Autonomic Thread Parallelism and Mapping Control for Software Transactional Memory

Naweiluo Zhou

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatics**

Arrêté ministérial : 7th August 2006

Présentée par

Naweiluo Zhou

Thèse dirigée par **Éric Rutten, Jean-François Méhaut**
et codirigée par **Gwenaël Delaval**

préparée au sein **Laboratoire d'informatique de Grenoble - LIG**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Autonomic Thread Parallelism and Mapping Control for Software Transactional Memory

Thèse soutenue publiquement le **19th October 2016**,
devant le jury composé de :

M.Raymond Namyst
Professeur, Université de Bordeaux, Rapporteur
M.Lionel Seinturier
Professeur, Université Lille 1, Rapporteur
M.Christian Perez
Directeur de recherche, INRIA Lyon , Examinateur
M.Jean-François Méhaut
Professeur, Université de Grenoble Alpes, Directeur de thèse
M.Éric Rutten
Chargé de recherche, INRIA Grenoble , Directeur de thèse
M.Gwenaël Delaval
Associate professeur, Université de Grenoble Alpes, Co-Directeur de thèse
M.Bogdan Robu
Associate professeur, Université de Grenoble Alpes, Invité

GRENOBLE ALPES UNIVERSITY
INRIA

Autonomic Thread Parallelism and Mapping Control for Software Transactional Memory

A THESIS
SUBMITTED TO GRENOBLE ALPES UNIVERSITY
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY (PHD)

Author:
Naweiluo ZHOU

Supervisors:
Prof. Jean-François MÉHAUT
Dr.Éric RUTTEN
Dr.Gwenaël DELAVAL

CORSE and Ctrl-A Research Teams
2016

Contents

List of Figures

List of Tables

Abstract

Parallel programs need to manage the trade-off between the time spent in synchronisation and computation. The trade-off is significantly affected by the number of active threads. High parallelism may decrease computing time while increase synchronisation cost. Furthermore, thread placement on different cores may impact on program performance, as the data access time can vary from one core to another due to intricacies of its underlying memory architecture. Therefore, the performance of a program can be improved by adjusting its parallelism degree and the mapping of its threads to physical cores. Alas, there is no universal rule to decide them for a program from an offline view, especially for a program with online behaviour variation. Moreover, offline tuning is less precise. This thesis presents work on dynamical management of parallelism and thread placement. It addresses multithread issues via Software Transactional Memory (STM). STM has emerged as a promising technique, which bypasses locks, to tackle synchronisation through transactions. Autonomic computing offers designers a framework of methods and techniques to build autonomic systems with well-mastered behaviours. Its key idea is to implement feedback control loops to design safe, efficient and predictable controllers, which enable monitoring and adjusting controlled systems dynamically while keeping overhead low. This dissertation proposes feedback control loops to automate management of threads at runtime and diminish program execution time.

Research Collaboration and Scientific Context

This PhD project is a sub-program under the Grenoble Project HPES (High Performance Embedded System). This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French Program Investissement d'avenir. This thesis joins work between High Performance Computing and Control Theory. It receives technique and technology support from two INRIA[1] research teams (CORSE and Ctrl-A) in Grenoble, France. The CORSE (Compiler optimisations and Runtime Systems) research team works toward both execution time optimisation and energy consumption improvement for HPC and embedded processors. Ctrl-A is a research team that majors in autonomic computing with its objectives to design safe controllers for automatic, adaptive, reconfigurable computing systems. The author is affiliated to both of the research teams. This work also collaborates with the control theory department of GIPSA-lab, which is a joint research unit of CNRS[2], Grenoble INP[3] and Grenoble Alpes University conducting theoretical and applied research in signals and systems. Thanks to the collaboration among the research teams and institutes, this work is able to show insights into the methodology of automatic control for high performance computing systems.

[1] French Institute for Research in Computer Science and Automation.
[2] French National Center for Scientific Research.
[3] Grenoble Institute of Technology, France.

Acknowledgements

The three years spent in Grenoble is one of the most joyful, peaceful time of my life. I must express my appreciation to all the people who have helped me through the years. I shall remember all the kindness.

I would like to take this opportunity to especially thank my PhD thesis supervisors Prof. Jean-François Méhaut, Dr. Éric Rutten and Dr. Gwenaël Delaval who have guided and supported me during my studies with their patience and knowledge whilst allowing me the space to work in my own way. I attribute my research to their constructive advices and efforts, without them this thesis would not have been completed otherwise.

I also would like to express my gratitude to Dr. Bogdan Robu who contributed tremendously to the development of the algorithms in my thesis and who has given all the supports to my publications.

The members of CORSE and CTRL-A research teams in INRIA have contributed immensely to my personal and professional time in Grenoble. The two groups have been a source of friendships as well as good advice and collaboration. I am especially grateful for my fun group members and friends Kevin Pouget, Brice Videau, Fabian Gruber and Christian Heinrich who have assisted me in my research and contributed greatly to proofread my dissertation. I acknowledge their solicitude and thoughtfulness.

Lastly, I offer my deepest gratitude to my parents who are always supporting me all through my life.

Chapter 1
Introduction

Chapter 1

Introduction

The last decades have seen a remarkable performance advancement of sequential processors with an annual increment rate of 40%-50% [1] in speed. This was achieved by doubling the amount of transistors on a single processor every two years to increase clock speed. However, this development is reaching an end due to physical limitations of transistors and obstacles of power dissipation among them. This hurdle promotes designs of multi-core processors. Rather than increasing clock frequency, a multi-core processor packs many cores that communicate through shared memory on one chip. Computation is accelerated by high concurrency. The performance benefits brought by multi-core processors require a program to execute in parallel and to scale when the number of cores increases. However writing a parallel application is difficult, as parallel programming encompasses all of the difficulties of sequential programming and introduces extra problems on coordination of interactions among concurrently executing tasks [1]. In addition, complex memory hierarchies are built on multi-core processors, which consist of several levels of cache to alleviate penalties of accessing the main memory. Consequently, parallel applications need to evolve to efficiently exploit potentials of their underlying multi-core platforms.

Multi-core processors boost program performance through high thread parallelism (number of simultaneously active threads). High parallelism shortens execution time, but it may also potentially increase synchronisation time. Furthermore, the complexity of the memory hierarchy causes diverse access latency from cores depending on the level of cache where data are placed. To alleviate access latency, threads can be fixed to certain cores to improve their resource usage, such as cache, main memory and interconnections. Therefore, the overall performance of a parallel application not only depends on its level of thread parallelism but also its thread localities on the cores.

The conventional way to address synchronisation is via locks. However, locks are notorious for various issues such as deadlocks as well as the vulnerability to failure and faults. Moreover, it is not straightforward to analyse interaction among concurrent operations. Transactional memory (TM) has emerged as an alternative parallel programming technique that handle synchronisation through transactions rather than locks. Access to shared data is enclosed in transactions that are executed speculatively without blocking by locks. Various TM schemes have been developed including Hardware Transactional Memory (HTM) [2], Software Transactional Memory (STM) [3] and Hybrid Transactional Memory (HyTM) [4]. This thesis presents the work on thread management under STM systems where the synchronisation time originates from transaction aborts. There are different ways to reduce the number of aborts, such as the design of contention manager policies (resolve conflicts among transactions), the way to detect conflicts (detect at early stage or later stage), the setting of version management (handles the storage policy for permanent and transient data copies) and the level of thread parallelism.

Online parallelism adaptation recently begins to receive attention. A suitable parallelism degree in a program can significantly impact on program performance. However, it is onerous to determine a parallelism degree for a program offline especially for the one with online behaviour variation. With regard to programs with online behaviour fluctuations, there is no single parallelism degree can enable its optimum performance. Therefore the natural solution consists of monitoring the program at runtime and altering its parallelism when necessary. Additionally, application performance is affected by diverse locations of threads. When the active thread number varies, locations that threads are pinned to may also need to be adjusted accordingly in order to optimise usage of the memory hierarchy. Pinning multiple threads to specific cores is called thread mapping [5] and specifying a thread to a specific core is addressed as thread affinity setting.

Furthermore, the diversity of TM applications and their supporting TM platforms together with the complexity of multi-core processor architecture make it difficult to decide the configurations of various parameters offline. Dynamical interactions among applications, TM platforms and underlying hardware can impact on system performance. All the aforementioned issues are preferred to be dealt with at runtime. Autonomic computing [6] is a technique that can automatically manage systems given high-level objectives. This thesis introduces feedback control loops into STM systems to achieve autonomic computing, more specifically, to automatically regulate thread

parallelism and mapping at runtime.

1.1 Contributions

This doctoral dissertation contributes to the areas of parallel systems and autonomic computing. It argues that online thread management is necessary and feasible for STM systems. It demonstrates that program performance is sensitive to thread parallelism and mapping. The contributions of this dissertation are as follows which have been partially published in [7, 8, 9, 10]:

1. Feedback control loops are employed to manage STM systems at runtime.
2. Two models are presented to detect near-optimum parallelism degrees for STM systems.
3. A model is proposed to coordinate adaptation of thread parallelism and mapping.
4. Two runtime phase detection algorithms are proposed and evaluated.

1.2 Thesis Outline

This thesis firstly present the relevant background in Chapter 2 to assist readers to better understand the remaining chapters. Chapter 2 begins with the description of multi-core processors and continues with the background on synchronisation mechanisms. This chapter later introduces the diverse benchmark applications utilised in this thesis for performance evaluation. Autonomic computing technique is presented lastly.

Chapter 3 proceeds to outline the overall proposed system architecture and the description of phase detection algorithms. The phase detection algorithms are utilised in the following three contribution chapters.

Chapter 4, Chapter 5 and Chapter 6 progressively detail the contributions of the thesis on thread parallelism control, thread mapping control and their coordination. Chapter 4 describes two approaches that can dynamically adjust parallelism degrees. Chapter 5 presents the method for managing thread localities. Chapter 6 gives coordination of parallelism and thread localities.

Related work is reviewed in Chapter 7 to describe the state of art and compare its significance with the contribution of this dissertation.

The last chapter (Chapter 8) gives conclusion remarks and proposes future work.

Chapter 2

Background

Chapter 2

Background

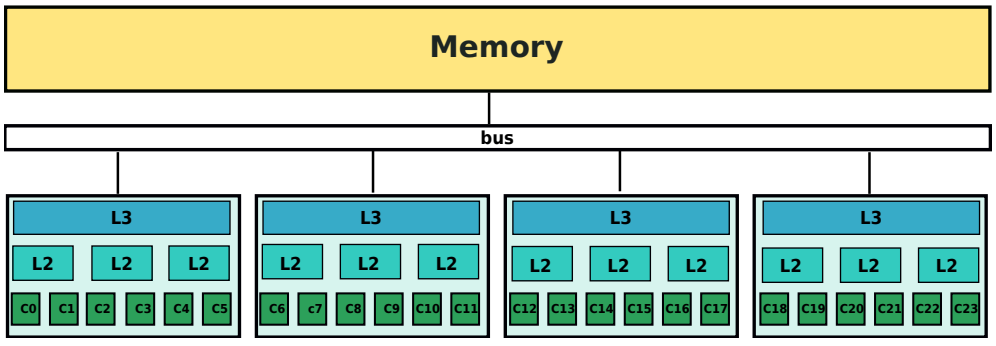
The performance of a parallel application can be significantly affected by: its thread parallelism degree, the location of threads on cores and the synchronisation mechanisms that are used to deal with concurrent access. This chapter describes significant historical background techniques and technologies to motivate and better understand the contributions of this dissertation. This chapter is organised as follows.

To start with, Section 2.1 describes the background information on modern multi-core processors which facilitates understanding of the rationales of performance impact from parallelism degree and thread affinity. Next, Section 2.2 reviews the main synchronisation mechanisms. The synchronisation techniques and technologies on Transactional Memory (TM) are described separately in the following section (Section 2.3). Then Section 2.4 introduces two benchmark suites that are widely used for performance evaluation on TM systems. Lastly, Section 2.5 outlines the necessary background on autonomic computing techniques, as autonomic computing plays a significant role in the methodologies that presented in this doctoral dissertation. Autonomic computing facilitates the Software Transactional Memory (STM) system in monitoring its runtime behaviour and responding to the changes accordingly.

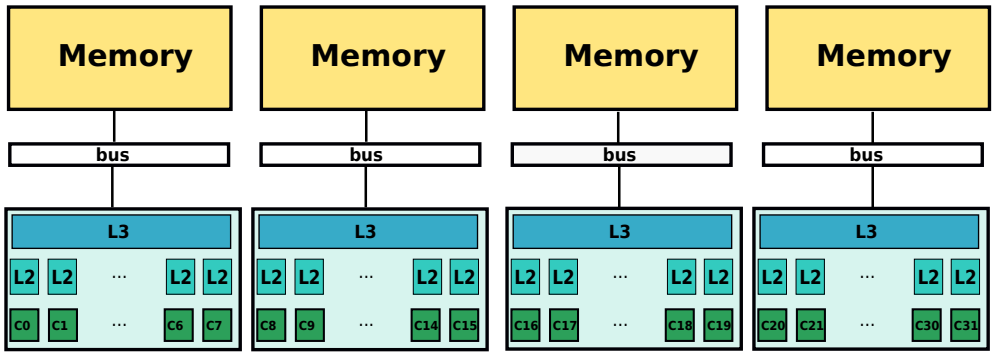
2.1 Multi-core Processors

Moore's Law [11] (doubling of transistors on a chip every 18 months) has been a fundamental driving force of processor designs. However, since 2005, processor designers have increased core counts to exploit Moore's Law scaling, rather than focusing on single-core performance [12], as the increase in CPU clock frequency is approaching a physical end. It is intricate to continue increasing computing speed of a single-core

processor based on current techniques and technologies, hence an alternative way to continuously improve performance for a high-end processor is to support many cores on one processor and multiple processors on one platform. A multi-core platform outstrips a single-core platform by scheduling multiple threads (or processes) for executing programs simultaneously. This somehow relieves the demands for high CPU clock frequency. Although structures of multi-core platforms can vary from manufacturers, they mainly fall into two classes [13]: *centralized shared-memory architecture* and *distributed memory architecture*. The first group is also known as *uniform memory access* (UMA), as all the processors have the equal access to one single centralized memory. The second group consists of multi-processors with distributed memory, known as *Non-Uniform Memory Access* (NUMA). Fig. 2.1 gives examples of UMA and NUMA platforms which are the topologies of the platforms utilised for performance evaluation in this thesis. The details of the platform configurations are provided in Table 2.1.



(a) An example of UMA topology



(b) An example of NUMA topology

Figure 2.1: Examples of UMA and NUMA platforms. Performance Evaluation is performed on the two machines.

Each socket of the UMA machine includes 6 cores and every two cores share a L2 cache, every 6 cores share a L3 cache and all cores share the main memory. Each

socket of the NUMA machine includes 8 cores. Each core has its own L2 cache and 8 cores share a L3 cache.

Characteristics	**UMA**	**NUMA**
Processor	Intel Xeon X7460	Intel Xeon Beckton X7560
number of cores	24	32
number of sockets	4	4
clock (GHz)	2.66	2.27
L1 cache capacity (KB)	32 (each)	32 (each)
L2 cache capacity (KB)	3072 (each)	256 (each)
L3 cache capacity(MB)	16 (each)	24 (each)
DRAM capacity (GB)	64	16 (each)

Table 2.1: Configurations for the UMA and NUMA platforms.

In both UMA and NUMA groups, as the distance between a core and a memory increases, the time to access data rises. More specifically, the time latency is lower to access L1 or L2 cache than that to the main memory. When it refers to a NUMA machine, the time latency is alleviated from a core to a remote memory (in contrast with its access to its local memory). The distributed memory in a NUMA machine is accessed via an *interconnect* by a core from its contiguous memory. Fig. 2.2 illustrates the access latency from the core to different memory levels[1]. Placing threads on the sibling cores which share all the levels of memory structure allows threads to reuse the data which already resides in the cache. This strategy can benefit the applications whose threads possess a significant amount of joint data access. Some applications showing disjoint data access may benefit from placement of distributed threads on cores, meaning that the threads do not always share the underlying memory structure, as it can alleviate potential contention.

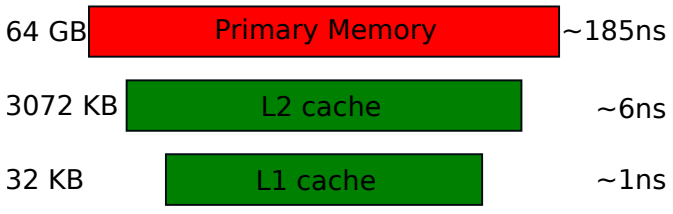


Figure 2.2: An illustration for access latency of diverse memory levels.

Most current processors are homogeneous both in instruction set architecture and

[1]**Lmbench** [14] is utilised to measure the latency of the UMA platform in Table 2.1. The L3 cache latency is skipped in the figured.

performance [15]. Some architecture, however, allow system software to control the clock frequency for each core individually in order to either save power or to temporarily boost single-thread performance. In contrast, heterogeneous architecture features at least two different kinds of cores that may differ in both the instruction set architecture, functionality and performance. The technique and technology on heterogeneous processors [16] are beyond the scope of the dissertation, the author is only concerned with homogeneous processors that keep clock frequency constant.

2.1.1 Thread affinity

Multi-core processors enhance program performance, *i.e.* reduce execution time, through high parallelism (number of simultaneous active threads). Furthermore, the complexity of modern memory hierarchies give diverse access latency from different cores, hence it becomes interesting to find the suitable tactic for thread placement in order to leverage resource usage. Applications that require significant interaction among threads, a small difference in thread management can give a non-trivial performance impact [17]. Assigning multiple threads to specific cores is called *thread mapping* [5] and fixing a thread to a specific core is called setting the *thread affinity*. In multi-core systems, there are three objectives in optimising thread placement [18]:

- **Better use of interconnects**. For instance, to reduce off-chip traffic by using intra-chip interconnects which have a higher bandwidth and lower latency.
- **Reducing invalidation misses**. A *invalidation miss* is when the data was already resident in the cache but is evicted by other cache lines. A common situation in shared-memory is to have one thread writing to an area of memory and another thread reading from the same location. This can cause one thread continuously to invalidate the cache lines of the other's. The objective of a thread mapping strategy is to reduce invalidation misses caused by two private caches holding the same data and continuously invalidating each other.
- **Reducing compulsory misses**. A *compulsory miss* (also known as *cold miss*) is caused by competition for the same cache. Threads will evict cache lines from each other if they share one cache, however, those cache lines would not be evicted if threads accessed the same group of addresses.

There are two means of tackling thread affinity [19] in most of operating systems. The first one, which is called soft affinity, relies on the traditional OS scheduler where

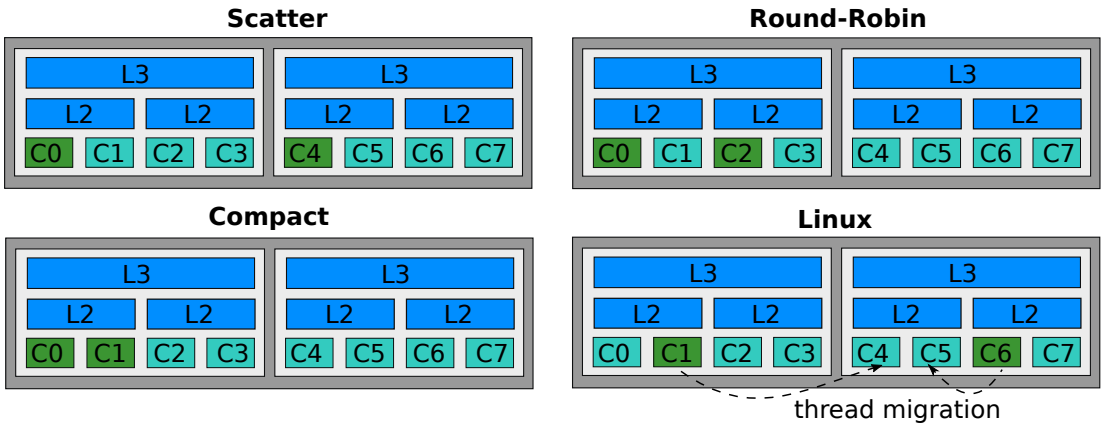


Figure 2.3: Four thread mapping strategies [5].

threads remain on the same core as long as possible. However, the OS scheduler may still migrate threads to another core or processor, possibly unnecessary, which impact on system performance. The second way of handling thread affinity is named hard affinity, meaning that allocation of threads is controlled by users. Depending on the number of cores and the memory hierarchy, there can be many different strategies to implement thread mapping. Using exhaustive search (which is utilised for optimisation on some conventional parallel program) by trying all the cases to figure out the best placement strategy is not feasible, especially when a large number of threads are involved. However if putting main memory aside, thread mapping strategies can be categorised into four main groups proposed by [5] as illustrated in Fig. 2.3:

- **Compact**: threads are placed on the sibling cores. This strategy is beneficial to applications in which the same amount of data is accessed by its threads. Threads sharing data but scheduled on the cores without sharing cache can result in excessive data movement and high network traffic [20].

- **Scatter**: threads are distributed across processors. Equally distributing threads is also addressed as *thread balancing* [21]. This strategy averts cache sharing among cores in order to reduce contention on the same cache. It is beneficial to applications whose thread mainly access disjoint data.

- **Round-Robin**: threads are placed on the cores where a higher level of cache (*e.g.* L3) is shared but not the lower level of cache (*e.g.* L2). This strategy can only be applied on the platforms which have more than one L3 shared cache and where the L2 cache is shared by more than one core.

- **Linux**: the default Linux scheduling strategy. It is based on dynamical priority-based strategy that allows threads to migrate to idle cores to balance the run queues. This mapping strategy corresponds to soft affinity, whereas the other three strategies coincide with hard affinity.

It causes thread migration to dynamically assign threads to specific cores. The most significant source of performance penalty when a thread migrates between cores is the loss of cache state [22]. The frequency of changing thread placement on cores is as important as the design of thread mapping strategies.

2.1.2 Memory Affinity

Memory affinity [23] is ensured when data is efficiently distributed over the machine memory. Such data distribution can either reduce the number of remote access or the memory contention. In the linux operating system, the default way to handle memory affinity is called *first-touch* [24], which places data on the node where is first accessed. In this dissertation, the author concerns little with direct manipulation of data placement in memory, but is more interested in investigating performance impact brought by thread affinity.

Multi-core processors bring parallel computing to the mainstream. Despite the potential performance boost by high concurrency, a parallel program is more difficult to write than a sequential program due to the necessity of synchronisation taking place among the concurrent tasks. There exists diverse techniques to address synchronisation issues as introduced in the following two sections.

2.2 Synchronisation Mechanisms

Multi-core platforms increase application performance by executing multiple threads concurrently. Threads must communicate and exchange data to complete their tasks, this is called *thread synchronisation*.

2.2.1 Lock-based Synchronisation Techniques

The conventional way to address synchronisation is through locks. Locks protect critical sections granting unique access by one CPU or one thread. The higher the parallelism is, the more time it may spend in contending for the lock. The time spent

in contending for the lock is called synchronisation cost. Lock-based synchronisation suffers from several limitations [25]:

1. **Deadlock**. This can cause the whole system to halt if circular dependencies exist where every job is stalled waiting for another job to complete.

2. **Long delay**. When one job which is the owner of a shared resource happens to be delayed, it causes all the other jobs which require the same resource access delayed.

3. **Priority inversion**. It happens when a higher priority task attempts to lock a mutex that is already locked by a lower priority task. Assuming P_1, P_2 and P_3 are low, medium and high priority tasks, respectively. If P_1 holds the mutex and P_3 has to wait for the release of the mutex from P_1. However if P_2 has been running before P_1 holds the mutex, P_2 will take precedence over P_3. The high priority task P_3 faces un unpredicted delay before it can run and it may miss its execution deadline.

The listed pitfalls result in the advent of other techniques to handle synchronisation issues as described in Section 2.2.2 and Section 2.3.

2.2.2 Other Synchronisation Techniques

One way to bypass locks is to utilise atomic primitives such as *CompareAndSet*. Non-blocking synchronisation algorithms are the ones that employ atomic primitives to achieve lock-free synchronisation.

Non-blocking synchronisation has its obvious advantages over lock-based synchronisation (or sometimes called blocking synchronisation) on the isolation between processes or threads accessing a shared object. This means that several processes or threads can operate on a critical section concurrently. An algorithm [25] is a *non-blocking algorithm* (sometimes called *lock-free*) if it guarantees that at least one process can complete a task or make progress within a finite time. In the literature, non-blocking synchronisation falls into three primary classes [26]:

- *Obstruction freedom*. It provides single-thread progress guarantees in the absence of conflicting operations. It does not rule out livelocks, as the threads may repeatedly preventing each other from making progress [27].

- *Lock freedom*. It provides system-wide progress guarantees. At least one process or thread is guaranteed to complete in a finite number of steps.

- *Wait freedom*. Every active process or thread can complete in a finite number of steps, regardless of the execution speeds on the others. It provides fault-tolerance [28], meaning that no process or thread can be prohibited from completing an operation by unknown halting failures of others, or by arbitrary fluctuations in their speed.

On one hand, non-blocking synchronisation mechanisms rely on atomic primitives, which can only operate on one word at a time resulting in complex structure for the algorithms. On the other hand, such mechanisms introduce extra cost to ensure that an object remains in one state where progress can be made even if the current process/thread dies. An alternative mechanism to tackle synchronisation issues is transactional memory, which overcomes the aforementioned drawbacks and shifts the burden of correct synchronisation from a programmer to a transactional memory system. Strictly speaking, transactional memory can be defined as a generic non-blocking synchronisation construct that allows correct sequential objects to be converted into concurrent objects [29].

2.3 Transactional Memory

Transactional memory (TM) emerges as an alternative parallel programming technique, which addresses synchronisation issues through transactions. The access to the shared data is enclosed in transactions that are executed speculatively without blocking by locks. Given that actual conflicts are rare in many applications [30], the TM approach is promising as the future programming model. The original idea of transactional memory dates back to 1977, when *Lomet et al.* [31] realised that an abstraction similar to a database transaction [32] might make a good programming language mechanism to ensure the consistency of data shared among several processes. Since then various TM schemes have been developed [2, 3, 4] including Hardware Transactional Memory (HTM), Software Transactional Memory (STM) and Hybrid Transactional Memory (HyTM).

2.3.1 Concepts of Transactional Memory

A transaction [33, 2] is a finite sequence of machine instructions, executed by a single process, satisfying the following properties:

- **Serializability**. The steps of one transaction never interleave with the steps of another.
- **Atomicity**. Each transaction makes a sequence of tentative changes to shared memory. When the transaction completes, it can either *commit*, making the changes to the memory permanent, or it can *abort* causing the previous changes to be discarded.

Tx management operations	Data access operations
void StartTx();	T ReadTx (T *addr);
bool CommitTx();	void WriteTx(T *addr,T v);
void AbortTx();	

Figure 2.4: The basic operations of TM

A transaction behaves as a logical unit, and its operations are either performed entirely or not performed at all. In other words, it behaves as if it were a single instruction. Failed transactions will be executed again. The stylised TM interface provides a set of operations for managing transactions and accessing data [34] without explicitly acquiring and releasing locks, as they are all handled by the TM implementation. Fig. 2.4 shows the basic operations for programming in TM. *StartTx* begins a new transaction in the current thread. So instead of acquiring a lock to access the critical section, in TM a *StartTx* instruction begins a task. *CommitTx* attempts to commit the current transaction but it may succeed or fail. Besides the commit operation, some systems also provide *AbortTx* which explicitly aborts the current transaction regardless of its conflict situation. The data access operations *ReadTx* takes the address *addr* of a value in type *T* and returns the transaction's view of the data at that address. *WriteTx*

```
1 do {
2     StartTx(); //start a new Tx
3     int tmp_x = ReadTx(&x); //read from x
4     int tmp_y = ReadTx(&y); //read from y
5     WriteTx(&y, tmp_x+tmp_y+1); write tmp_x+tmp_y+1 to y
6 } while (!CommitTx()); //commit the current transaction
```

Figure 2.5: A simple example on usage of transaction primitives [34].

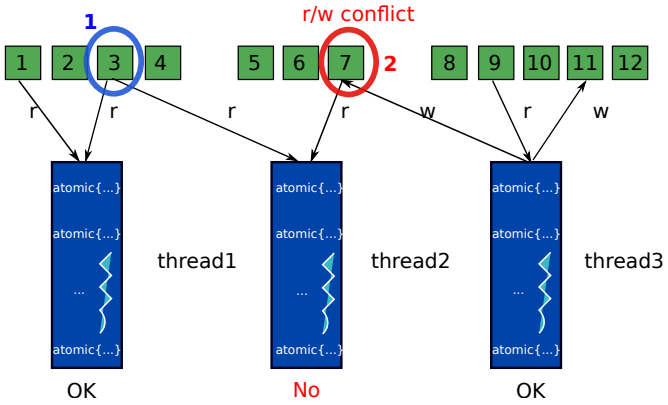


Figure 2.6: An example of transaction conflicts in TM. Object 3 and object 7 are accessing concurrently by different threads. A read and write conflict happens between thread 2 and thread 3.

takes an address *addr* and a new value *V*, writing the value to the transaction's view of that address. The data value type *T* differs from one system to another. With TM, the programmer specifies intents rather than mechanisms leading to a higher-level abstraction than locks. For instance, the programmer can concentrate on the decisions of the placement of the atomicity, in preference to the mechanisms that are used to enforce the intent [35]. Fig. 2.5 illustrates a simple example on usage of the transactional primitives of Fig. 2.4.

Conflicts can happen during a transaction execution or when it is committing. Fig. 2.6 shows an example of conflicting transactions. In the example, each thread executes a sequence of transactions. A read and write conflict happens between thread2 and thread3 when they are accessing the shared object7 concurrently. Object3 is read concurrently by two threads. Two read operations do not cause any aborts. TM employs various synchronisation mechanisms to mediate the read/write or write/write conflicts. Broadly speaking, there are two approaches for conflict control: pessimistic concurrency control and optimistic concurrency control. *Pessimistic concurrency control* detects and solves the conflicts at the same time when a transaction is about to access a location. This type of conflict control grants the data ownership to a transaction therefore preventing other transactions from accessing the same data. In *optimistic concurrency control*, the conflicts detection and resolution are delayed after data access therefore allowing multiple transactions to access the same data. When conflicts are frequent, the pessimistic concurrency control is favoured, as it enables a transaction to complete all of its operations once the access permission is granted. However, if the conflicts are rare, pessimistic concurrency control is inferior to optimistic concurrency

control as the later one increments the concurrency level by avoiding the locks.

TM provides serializability and atomicity. Hardware transactional memory rarely requires locking mechanisms, yet locks can not be completely removed especially for software transactional memory [36]. Since conflicts can not be avoided, different design choices have been designed aiming to diminish conflicts which are mainly based on: when to detect a conflict, how to solve a conflict and how to manage memory log.

2.3.2 TM Design Choices

There are four key aspects for TM designs: **granularity**, **conflict detection**, **conflict resolution** and **version management**. Every aspect includes multiple design choices which are either fixed for a TM system or can be selected by users before application execution. In some cases, it is possible to switch design choices at runtime. It is worth noting that the two approaches for conflict control (pessimistic and optimistic concurrency control) introduced in Section 2.3.1 are composed of conflict detection policies and conflict resolution policy. This section will unroll more details on that.

Granularity

Transaction granularity [34] is the unit of storage in which a TM system detects conflicts. The choice of granularity impacts on TM performance. Similar to the cache granularity, a fine granularity reduces false conflicts while a coarse granularity reduces overhead. HTM often uses cache-line granularity, so that the conflict is detected when a transaction is trying to change the status of a cache line. While STM operates on word or block granularity. Some TM systems employ object granularity which extends transactional protection to an entire object at once.

Conflict Detection Policy

Conflict detection [37] dictates when to check the read/write sets to detect conflicts. Two major designs are often employed, namely *eager* and *lazy*. The eager option detects conflicts for every memory access. The lazy option detects conflicts when a commit is required. The former option observes conflicts at early stage thus obviating large-size abort, but it can potentially impose high abort rate. The later option, on the other hand, delays the abort time which reduces the abort rate, however, it causes large-size aborts.

Contention Manager

The *contention manager* (CM) decides the actions to be taken in order to resolve conflicts. When a transaction encounters a conflict with another transactions, three possible decisions can be made [37]:

- *abort-other*: also known as *suicide*, when a transaction encounters a conflict with other transactions, it will kill the others which are in conflicts with so that its own data validation can be guaranteed.

- *abort-itself*: also known as *aggressive*, when a transaction detects conflicts with other transactions, it will kill itself to ensure the data validation of other transactions. This policy performs well on a lowly-contended application, but scales badly on highly-contended application as the aborted transactions are restarted right after the conflict is detected and are doomed to abort several times before a successful commit [5].

- *backoff*: when a transaction meets a conflict with other transactions (1) instead of aborting itself immediately, it stalls for a certain period of time and rechecks its data validation once it resumes; (2) it aborts immediately and stalls for a certain period before its re-execution.

Various conflict resolutions are proposed based on the above three schemes aiming to: minimise wasted work (aborts), avoid future conflicts and reduce the overhead of executing CM itself.

Version Management

Version management [34] handles the storage policy for permanent and transient data copies. The policy can be *eager* (also called write-throughput) or *lazy* (also known as write-back). Eager policy logs the old data and replaces the memory with the new data. Lazy policy logs new data and keeps the old data in memory. The former policy has lower commit-time overhead hence it is preferred for tackling read-after-write or write-after-write operations. The latter policy has lower abort overhead and does not require extra work to guarantee consistent reads.

2.3.3 TM Metrics

Different metrics in TM can be utilised to indicate the characteristics of TM applications, *Ansari et al.* [38] summarised the commonly used ones and proposed two additional new metrics as follows:

- **speedup**. This metric is intuitive. It depends on characteristics of both the application and the TM implementation.
- **in transaction**. It is the percentage of total time that the applications spent in transactions. Noting that non-transactional code also exists in a TM application. This metric, however, does not indicate application performance.
- **wasted work**. The aborted transactions. It is often calculated by dividing the total time spent in the aborted transactions by the time spent in all transactions. Or some works [39, 40, 41] are in favour of addressing wasted work as the total time spent in aborted transactions.
- **aborts per commit**. It indicates the mean aborted transactions per committed transactions.
- **abort histograms**. It details how abort per commit is spread among the transactions.
- **contention management time**. It is the percentage of time the mean committed transaction spends in performing contention management.
- **transaction execution time histograms**. It shows the spread of execution times of committed transactions. This metric is useful as it illustrates how homogeneous or heterogeneous the amount of work contained in transactions for a given application is. For instance, an application is more homogeneous with all its transaction executing the same code block, and less so when its transactions executing a group of code blocks.
- **instantaneous commit rate**. It shows the proportion of committed transactions at sample points during the execution of the application. Note that active transactions are not taken into account.
- **readset & writeset** . Size of read and write operations of a transaction. It facilitates the selection of buffer size or cache size for HTM. A *readset* of a transaction

is the set of locations read by the transaction. A *writeset* of a transaction is a set of locations accessed by the transaction.

- **readset-to-writeset ratio**. It indicates the mean number of reads lead to a write in a committed transaction.
- **writes-to-writeset ratio**. The mean number of writes to a transaction data element.
- **reads-to-readset ratio**. The mean number of reads to a transactional data element.

Additionally, two metrics are useful to indicate the online TM application performance, namely *commit ratio* (CR) and *throughput*. CR equals the number of commits divided by the number of commits and aborts; it measures the level of conflict or contention among the current transactions. Throughput is the number of commits in one unit of time; it directly indicates transaction progress rate. The two metrics will be discussed in more details in Chapter 3.

2.3.4 Implementation Schemes

Diverse implementation schemes have been developed [2, 3, 4] including Hardware Transactional Memory (HTM), Software Transactional Memory (STM) and Hybrid Transactional Memory (HyTM).

Software Transactional Memory

A Software Transactional Memory (STM) system implements all its transactional semantics in software. It provides a lock-free programming interface, however, the STM system itself is not necessarily lock-free. STM requires a sequence of locks to manage concurrent access to shared data. Contrasting with a lock-protected critical section which hinders concurrent access by multiple threads, STM locks are used to indicate the ownership by certain transactions [3]. This information is used later to detect conflicts when a transaction tries to commit. A programmer neither needs to indicate where to acquire/release locks nor to identify which operations may be allowed to execute concurrently, as this responsibility is taken by the TM implementation. The old data are logged and recovered once an abort takes place. Since both the old and new data are stored, there is a high demand for memory storage. When a transaction needs

to commit, it has to traverse its previous accessed locations to ensure that the data read previously has not been changed. Implementation of such a functionality purely in software delivers a high runtime cost making STM performance-wise inefficient for general purpose parallel programs. STM systems present worse performance than locks when a small number of threads are used. In contrast, in some cases where there is a high concurrency level with rare conflicts, STM systems manifest comparable or even better performance than conventional locks [1]. As when the number of threads rises, the cost of contending for locks rises accordingly.

This dissertation is concerned with STM systems as they are flexible (*e.g.* it is easy to modify their conflict control polices and they have straightforward interfaces) and rely little on underlying specific TM hardware supports. A brief description of state-of-the-art STM systems will be given later in Section 2.3.5.

Hardware Transactional Memory

Hardware Transactional Memory (HTM) implements its all transactional functionalities in hardware. A HTM system starts a transaction by executing a register checkpoint with shadow register files [42]. A *checkpoint* [43] is a program location in a transaction where control may jump during a partial abort. The cache coherence protocols of processors are modified to be compatible with transactional execution. A processor preserves two caches: one regular cache for non-transactional access and one transactional cache for transactional access. The transactional cache does not propagate the tentative writes to other processors or write to the main memory unless otherwise a transaction commits. A commit makes tentative writes visible to other processors and writes back the updates to the main memory. A conflict is detected by comparing the readsets and writesets. Once a conflict occurs, at least one of the transactions aborts and restores the values saved at the checkpoint at the start of the transaction to the registers. Meanwhile the previous tentative writes are dropped by the transactional cache.

Intel has introduced hardware support (Intel TSX) for TM in the Intel 4th Generation core ™ processors [44] in 2012. L1 data cache tracks transactional states in the granularity of a cache line and detect conflicts through cache coherence protocol. Intel TSX provides developers with two software interfaces to specify the critical sections, namely hardware lock elision (HLE) and restricted transactional memory (RTM). In HLE, an abort may lead a transaction to fall back to a lock-based execution to avert successive aborts which hinders the program progress. RTM requires a programmer

to provide a software handler to tackle transaction aborts. This allows the possibility of other strategies to solve aborts rather than giving up immediately on hardware transactions [45].

HTM systems, as all the functionalities are implemented into hardware, indicate a lower overhead than STM systems thus faster execution. Just as the initial intention of TM design, HTM systems do not necessarily require locking mechanisms. Nevertheless, HTM imposes restraints on the resource capacity, such as cache capacity. Furthermore, HTM offers limited scope for contention management which plays an important role in forward progress, thus persisting aborts exist in HTM. A common solution to this restraint is to grant the transaction a lock to ensure an exclusive execution, yet acquiring a lock by a transaction causes other transactions to abort. This can lead to a chain effect known as *lemming effect* where the other aborted transactions endeavour to obtain the lock [46]. An alternative solution to gain good performance as well as flexible size of atomicity is to combine HTM with STM.

Hybrid Transactional Memory

The construction of Hybrid TM (HyTM) system combines the base STM with a generic HTM system, thus removing part of high cost imposed by software data structure and also increasing the resource capacity limited by HTM. A transaction starts and progresses in the HTM mode and only enters the STM mode when a transaction aborts a certain number of times. Some tactics can be employed to reduce the abort rate reckoning on the causes of aborts [47]. An abort caused by hardware resource constraints will be executed in the STM mode as retrying in a HTM will still lead to persisting aborts. An abort due to conflicts can be re-executed in the HTM mode repeatedly.

Since both HTM and STM modes exist in one TM system, the additional complexity is added due to the intricacy of detecting the conflicts between a HTM mode transaction and a STM mode transaction [5]. *Per contra*, it is reported that HyTM can negatively affect some desirable properties of the original STM, especially the fairness of resource access [36].

2.3.5 Software Transactional Memory Platforms

Section 2.3.4 has briefly discussed the techniques of STM which mainly concentrates on the comparison of its difference with HTM and HyTM. This section presents implementations of different STM systems and compare their differences. Particularly,

TinySTM, which is reviewed in the end of the section, is utilised in the thesis as the STM platform. STM systems can be implemented in a library or directly into a compiler. A library-based approach relies on the programmer to convert sequential code or lock-based code into transactional code. A compiler or preprocessor can be used to enable automatic transformation. STM systems can also be classified by its programming languages (*e.g.* C, C++, Java, C#, Python, Perl, Haskell, Scala) and its granularities of data sharing (word-based or object-based). Despite the diversity, the following STM systems share some common features:

- *global timestamp*. It is logic timestamp, which is incremented by one per commit or write operation. The transaction checks the timestamp of its previously accessed objects upon its commit. A transaction can successfully commit if the timestamp of the previous accessed objects stay consistent.

- *array of locks*. Memory locations are mapped to locks based on certain functions. Locks are utilised to manage concurrent access to memory locations rather than stalling threads. The content of a lock can be either an address of an owner transaction or a timestamp (details are presented in the following sections).

TL2

Transactional locking II (TL2) algorithm [48] is the second version of the original transactional locking (TL) proposed by *Dice et al.*. The TL2 algorithm introduces a global version-lock that is augmented by each writing transaction at commit time. Each thread has a local variable to load the current value of the global version number. Every transactional memory location is associated with a write-lock. A lock contains a bit indicating whether the lock is taken and the rest of the bits indicating a version number. The version number is the timestamp.

TL2 employs a two-phase locking scheme. A transaction first obtains the current value of the global version clock (called read version number) and stores it in a thread local variable. Every load operation is followed by a validation operation, which checks if the location's write lock is free (to check if it is conflicting with other transactions) and has not been changed. Additionally, the lock's write version number (a version number associated by a write lock) should be no greater than the transaction's read version number. Otherwise the transaction aborts, as the memory location has been modified after the current thread performs the load operation. All the transactions are executed speculatively and conduct no change to the shared memory states

before committing. At the end of the speculative execution, all the previous performed operations of one transaction need to be revalidated. This is achieved by comparing the read version number with the lock's version number. When the version number of the write lock is greater than the read version number, the transaction is aborted. The new version number is recorded in a local variable. At commit time, the new value should be stored to the memory location and the locations locked should be released. One exception is when a read version number plus one equals a write version number, validation of the read transaction can be skipped.

To reduce the overhead the global version number is split into two parts: one part for the version number, and one part for the thread id (which records the last thread that updated it). A thread does not need to change the version number if it detects the global version number differs from its version number. A thread only performs an update (increment by one) to the global version number when its own version number equals the global one.

SwissTM

SwissTM [49] is a lock-based STM which utilises word-based granularity to map memory locations to locks. It employs a global commit counter (timestamp) which is incremented by every non-read only transaction and used for transaction validation upon commits. Every transaction starts by reading the global commit counter. When the transaction *reads* a memory location, it first reads the write lock. If the transaction does not own the current write lock, it reads the read lock and reads the read lock again after its reading operation. The read operation is validated if the values of the read lock at the two consecutive reads are the same. Upon the successful validation, the transaction extends its validation timestamp to the value of the current global commit counter. If validation fails, the transaction rolls back. When the transaction *writes* to a memory location, it also first checks if it owns the write lock. The transaction updates the value in the memory location directly if it already owns the lock, otherwise the transaction tries to obtain the write lock. Failing to obtain the write lock leads to intervention of the contention manager which decides how to resolve the conflict. Upon commit, a read-only transaction can commit immediately. A transaction with write operations has to validate all its previous read operations to guarantee their consistence with the current state of the values. Upon successful validation, the transaction updates the memory location and releases the read and write locks as well as incrementing the global commit counter.

SwissTM employs a conflict detection scheme that treat transactions differently from TL2. It detects write/write conflict eagerly to prevent transactions that are doomed to abort from further wasting resource; it detects read/write conflicts lazily to allow more parallelism. A two-phase contention manager is employed in SwissTM which incurs no overhead on read-only and short read-write transactions and allows the transactions which have performed a significant number of updates to progress. Each transaction records its write access which indicates the phase. The first phase marks the number of writes less than a threshold. A transaction aborts and restarts immediately once a conflict is met during the first phase. When exceeding this threshold, transactions are in the second phase where they abort and back-off. In addition, the transactions that abort due to write/write conflicts back-off for a period which is proportional to the number of the successive aborts.

RSTM

Rochester Software Transactional Memory (RSTM) [50] is one of the oldest open-source STM systems. RSTM is a C++ library for object-oriented transactional programming. It includes several STM algorithms that allow it to be customised to suit a given workload. The implementations of the algorithms can differ in: bookkeeping, visible reader management and memory management.

There are two major types of bookkeeping strategies, *i.e.* DSTM [27] and OSTM [51]. DSTM dynamically initialises and deletes the transaction locator (the shared data) which is accessed by transactions. The locator contains the information of the latest transaction which has written to it, the new data version and the old data version. The transaction utilises an object header (a pointer) to points to the locator so that the contents of the locator can be modified atomically. While in OSTM, each transaction maintains a transaction descriptor which records a list of the shared read and a list of shared write objects. The lists contain object handles which refer to the object header (a pointer which points to the current object). Each time when a transaction needs to access an object, it accesses an object header. In contrast, RSTM mixes the above two strategies, rather than utilising a locator, RSTM merges its contents into the new data object which also points to the old data object. Each transaction also owns a transaction descriptor which only contains the status of the transaction (in contrast with OSTM's descriptor, which also maintains two lists of read and write objects). When a transaction demands access to an object, it accesses an object header which points to the share objects. The object header also maintains a list of visible reads which

serves to avoid the cost of validating invisible reads. The object header and transaction descriptor of RSTM require no dynamic memory allocation.

TinySTM

TinySTM [3] is a word-based (granularity) lightweight STM system which employs a shared array of locks to control concurrent access to memory and applies a shared counter as clock to indicate the global timestamp. The role of locks in TinySTM is to enable atomicity when hardware support is missing, rather than locking memory locations, therefore its locks do not intercept memory access by multiple threads.

A TinySTM lock is the size of an address with its least significant bit indicating whether the lock is owned by a transaction. If not owned, the remaining bits of the lock store the version number (timestamp) of the transaction which is the last to write to one of the memory locations covered by the lock. If it is owned, the remaining bits store an address to either the owner transaction (when using write-through version management) or to an entry in the write set of the owner transaction (write-back version management). When the least significant bit is set, a writing transaction first checks if it is the owner of the lock, if not, the transaction can write the new value directly otherwise it has to wait for a certain period or abort immediately. If the least significant bit is not set, the writing transaction tries to acquire the lock by setting this bit using an atomic operation (*e.g.* Compare and Swap). A failure indicates that another transaction has acquired the lock concurrently. A reading operation by a transaction verifies if the lock is owned or updated concurrently. A reading operation is consistent if the lock is not owned and its value has not been changed between both read operations.

At commit time, a transaction needs to verify its previous read operations. The verification process can be costly. TinySTM employs hierarchical locking to diminish the cost of validating a large chunk of memory. In addition to the shared array of locks, a smaller hierarchical array of counters (different from the global counter which is used as the timestamp) are maintained. Multiple locks are mapped to one counter using a hash function. Transactions additionally maintain a read mask and write mask specifically for the counters. A read operation sets its read mask and stores the value of the counter, and a write operation sets its write mask and increments the counter. Upon validation, the counters with corresponding read masks set are checked if: the value of the counter is equal to the stored value in the transaction, or the counter is one more than the stored value when the write mask is set. If true, the corresponding locks traverse can be skipped. Comparing the locks in TL2, TinySTM does not require

storing of the thread id and it provides an additional hierarchical locking layer to gain performance for read operation validation.

TinySTM provides memory-management functions that allow transactional code to use dynamic memory. Transactions keep track of memory allocated or freed: allocated memory is automatically disposed of upon abort, and freed memory is not disposed of until commit. Furthermore, a transaction can only free the memory locations after it has acquired all the locks covering the memory locations.

TinySTM provides interfaces allowing users to choose different parameters easily, such as conflict detection policies, version management policies and CM policies. Additionally, it also grants easy interface to user-defined functions.

2.3.6 Restrictions of STM

Despite of its merits, a STM system is complicated. It incorporates numerous tunable parameters such as contention manger, version management policy and so forth. Such tunable parameters are usually set prior to application execution and remain consistent during the whole program execution. Few actions from the STM can be made to adapt the system to the diversity of program runtime behaviour. Alongside the underlying multi-core processors and divergence of the applications, the intricacy is further enhanced. Manual offline tuning is arduous and less precise. *Autonomic computing* (Section 2.5) is capable of monitoring the behaviour of applications and STM systems at runtime and tuning their parameters accordingly to ameliorate performance. Before starting to investigate autonomic computing techniques, it is necessary to present several benchmark applications that are widely used to evaluate performance of TM systems in the next section.

2.4 Benchmarks for Evaluation of TM Systems

Since the advent of the first TM proposal, efforts have been seen to develop TM microbenchmarks and benchmarks to evaluate performance of TM implementations. This section gives a brief introduction to the benchmark suites that are utilised in the dissertation: an artificial benchmark suite (**EigenBench**) and a realistic benchmark suite (**STAMP**). There are other available TM benchmarks [52, 53, 54, 55] which are beyond the scope of the thesis.

2.4.1 EigenBench

EigenBench [56] is an artificial but highly configurable benchmark suite. By adjusting the input parameters, **EigenBench** can be easily configured to yield the workloads of concern for TM evaluation. In addition, it can be configured to mimic the behaviour of real TM applications as illustrated in [56]. The core of **EigenBench** is three separate arrays whose sizes can be tuned. *Array1* is the *hot* array that is shared by all the threads and accessed only by transactions. *Array2* is the *mild* array that is accessed by transactions but whose data is private to each thread. *Array3* is the *cold* array that is accessed by non-transactions and private to each thread. The kernel of **EigenBench** code is presented in Fig. 2.7. The parameter *loops* stands for the number of transactions assigned to each thread. Tuning *R1*, *R2*, *R3*, *W1*, *W2* and *W3* can change the number of read, write operations inside transactions. Moreover, it is also possible to configure the size of non-transaction operations. *random_actions* function determines whether a transaction performs a read or write and whether to access the hot or mild array. *local_ops* conducts a given number of reads or writes on the cold array and performs *nops*.

```
1  void test_core(tid, loops, pesist, lct, R1, W1, R2, W2
2                R3_i, W3_i, Nop_i, k_i, R3_o, W3_o, Nop_o, k_o)
3  {
4   long val=0;
5   long total = W1 + W2 + R1 + R2;
6   for (i=0; i<loops; i++)
7      {
8        Save_Random_Seed;
9        BEGIN_TM();
10       if (persist) Restore_Random_Seed;
11           (r1,r2,w1,w2) = (R1,R2,W1,W2);
12            Reset_History_Buffers;
13       for (j=0; j<total ; j++)
14          {
15           (action, array) = rand_action(r1, w2, r2, w2);
16           index = rand_index(tid, lct, array);
17           if (action == READ)
18               val += TM_READ(array[index]);
19           else
20               TM_WRITE(array[index], val);
21           if ((j%k_i)==0)
22              val += local_ops(R3_i, W3_i, Nop_i, val, tid);
23         }
24       END_TM();
25       if ((i%k_o)==0)
26           val += local_ops(R3_o, W3_o, Nop_o, val, tid);
27     }
28 }
```

Figure 2.7: PseudoCode description of **EigenBench**'s core code [56].

2.4.2 STAMP

Stanford Transactional Applications for Multi-Processing (STAMP) [57] is a benchmark suite which incorporates 8 applications. Inside the applications, there are 30 variants of input parameters and data sets provided. The applications simulate a wide range of transactional behaviours, such as size of transactions, amount of contention, size of read and write operations, coarse-grain and fine-grain transactions. Additionally, they are compatible to HTM, STM and HyTM designs. The characteristics of STAMP are summarised as follows:

- **bayes**. This application implements an algorithm for automatically learning structures of Bayesian networks from observed data. The Bayesian network is represented as a directed acyclic graph, whose nodes represent variables and whose edges represent conditional dependencies among variables. Initially, there is no dependencies among the variables. Its algorithm continuously learns dependencies by analysing the observed data as the application progresses. A transaction is utilised to protect the calculation and addition of a new dependency. Overall, this application has long transaction length (the time spent in committing one transaction), high transaction time (execution time of transactions) and high contention. However, **bayes** exhibits non-determinism [58]: the ordering of commits among threads at the beginning of its execution can drastically impact on execution time.

- **genome**. This application performs a process of taking a significant number of DNA segments and matching them to reconstruct the original source of genome. More specifically, the process is composed of two phases. The first phase utilises a hash set to create a set of unique segments. In the second phrase, each thread attempts to remove a segment from a global pool of unmatched segments and adds it to its partition of currently matched segments. Transactions are embedded in each phase which evades the implementation of a deadlock avoidance scheme. Overall, this benchmark has medium transaction length, high transaction time and low contention.

- **intruder**. This application is a signature-based network intrusion detection system (NIDS), which scans network packets for matches against a known set of intrusion signatures. Network packets are processed in three phases: capture, reassembly and detection, with the latter two phases enclosed by transactions.

Overall, this benchmark has short transaction length, medium transaction time and high contention.

- **kmeans**. This application is commonly applied to partition data into related subsets. Particularly, it groups objects in N-dimensional space into K clusters. Each partition of the objects is processed by one thread with transactions wrapping the update of the clusters. The intensity of contention depends on the value of K and the size of the transactions are proportional to the dimensionality of the space. Overall, this benchmark has short transaction length, low transaction time and low contention.

- **labyrinth**. This application implements a variant of Lee's algorithm [59]. The main data structure is a maze which is implemented by a three-dimensional uniform grid. The calculation of the maze's path is enclosed by a single transaction. A conflict happens when two threads try to fetch paths that overlap. Transactions are efficient in this benchmark as they obviate the deadlock avoidance schemes. Overall, this benchmark has long transaction length, high transaction time and high contention.

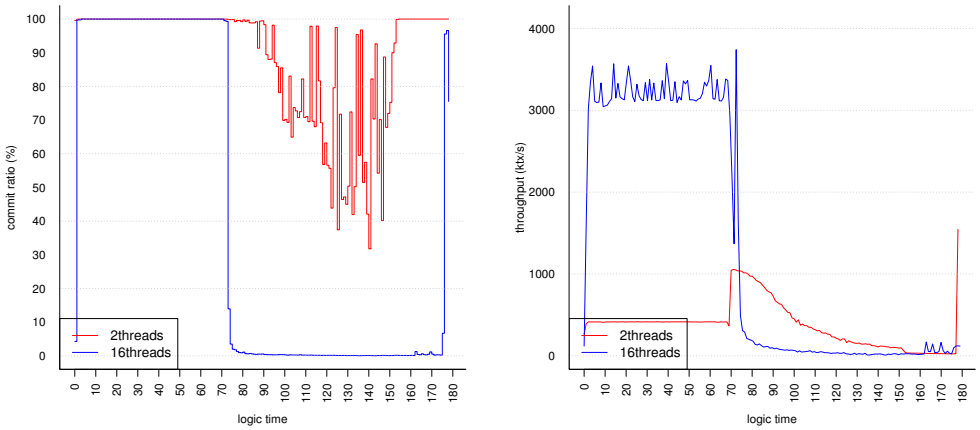
- **ssca2**. It stands for Scalable Synthetic Compact Applications 2 (SSCA2) which contains 4 kernels that operate on a large, directed, weighted multi-graph. STAMP only adopts Kernel 1, *i.e.* constructing an efficient graph data structure by adjacent and auxiliary arrays. Transactions are utilised to protect accessing the two arrays. Overall, this benchmark has short transaction length, low transaction time and low contention.

- **vacation** This application implements an online application transaction system which emulates a travel reservation system. More specifically, it implements a set of tree data structures that continuously track customers and their reservation for various travel items. Threads conduct a number of sessions which interact with the system database. Each of the sessions are enclosed into transactions resulting in a significant amount of time spent in transactions. Overall, this benchmark has medium transaction length, high transaction time and low/medium contention.

- **yada**. YADA is the abbreviation of Yet Another Delaunay Application. It implements Ruppert's algorithm for Delaunay mesh refinements [60]. The basic data structure of **yada** is a graph which stores mesh triangles, a set of mesh boundary

segments and a task queue. The algorithm executes by iterating the following steps: a triangle is removed from the queue, its retriangulation is performed on the mesh and the new triangles obtained from the retriangulation are added to the queue. Access to the queue is enclosed by transactions. Overall, this benchmark has long transaction length, high transaction time and medium contention.

Some of the aforementioned applications demonstrate behaviour variation at runtime. Such online performance fluctuations can be directly reflected on contention changes. For instance, **genome** possesses three phases as shown in Fig. 2.8. A very short phase with medium contention followed by a relatively long phase where the contention is zero, as all the operations are reads, the last phase is characterised by a rise of write operations leading to high contention. The contention falls again when the program is approaching the end, as a part of threads are accomplishing their operations resulting in less conflicts globally. Fig. 2.8(a) illustrates contention fluctuations for the parallelism degree 2 and 16. The corresponding throughputs are indicated in Fig. 2.8(b). Fig. 2.8 demonstrates that varying parallelism degrees can impact on application contention, therefore possibly improve application performance on execution time. The next section describes the background technology and technique on autonomic computing which is capable of controlling application behaviour automatically.



(a) Contention (b) Throughput

Figure 2.8: Online variation of contention and throughput for **genome**

2.5 Control of Autonomic Computing Systems

Autonomic Computing enables a software system to be self-adaptive. This technique is introduced to STM systems in the thesis so that the systems is able to improve their performance automatically.

2.5.1 Concepts of Autonomic Computing

The term "autonomic" comes from biology [61]. In the human body, the autonomic nervous system takes care of unconscious reflexes without which, humans would have to be constantly busy consciously adjusting the body to its needs and the environment. IBM gives the definition of *autonomic computing* [62, 6]– computing systems that can manage themselves given high level objectives from administrators.

A system can be considered as an autonomic system if it incorporates one of the following aspects:

1. **self-optimisation**. Some complex systems may include multiple tunable parameters, which require careful tuning to maximise performance. In such a system, the components and the system itself should be able to seek opportunities to improve its own performance efficiency and throughput automatically by adjusting these parameters. Such tunable parameters are also addressed as *control points* [63, 64] in some literature.

2. **self-configuration**. The system can configure itself automatically in accordance with high-level policies meaning that when a new component is added to a system, it can incorporate itself seamlessly and the rest of the system can adjust to its presence.

3. **self-healing**. The system can detect, diagnose and repair the problems deriving from bugs or failures.

4. **self-protection**. Firstly, the system can defend itself from malicious attacks or cascading failures that are uncorrected by self-healing measures. Furthermore, the system can anticipate problems based on the information collected by sensors, so that these problems can be avoided or mitigated.

This dissertation concentrates on the first feature: **self-optimisation**.

Autonomic computing proposes a general structure of feedback loop to take adaptive and reconfigurable computing into account [65]. Feedback control loops, on one

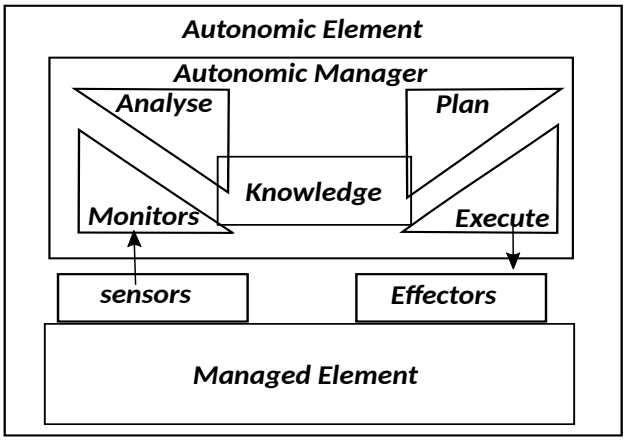


Figure 2.9: A MAPE-K control loop. It incorporates an autonomic manager, a sensor, an effector and a managed element among which the autonomic manager plays the main role.

form or another, have been adopted as cornerstones of software-intensive self-adaptive systems [66]. A classic MAPE-K feedback control loop proposed by IBM is introduced in the following section.

2.5.2 MAPE-K Loop

Feedback control loops provide a generic mechanism for self-adaptation which is an essential requirement for computing systems. However, in software engineering, feedback control loops are often hidden, abstracted, dispersed or internalised when the architecture of an adaptive system is documented or presented [67]. Increasing the visibility of feedback control facilitates the uncertainty management of computing systems [68]. A classic feedback control loop is illustrated in Fig. 2.9 in the shape of a MAPE-K loop (Monitor, Analyse, Plan, Execute, Knowledge) proposed by IBM.

In general, a feedback control loop (also called an autonomic element) possesses:

1. **autonomic manager**. It is also known as a controller.

2. **sensor**. It is also called probe or gauge which collects information from the managed element. *E.g.* on a web server, the sensors can be the response time to client requests, network and CPU usage, CPU and memory utilisation.

3. **actuator**. It carries out changes to the managed element. *E.g.* adding or removing machines to a web server.

4. **managed element**. It can be any software or/and hardware that is given autonomic behaviour by coupling it with an autonomic manager.

An autonomic manager takes the information of concern from the sensor to *monitors* and execute changes via *actuators*. An autonomic manager is composed of five elements:

1. **monitor**. It is used for sampling. The autonomic manager needs the appropriate monitored data to recognise failure or suboptimal performance of the autonomic element and perform the appropriate actions. There are two types of monitoring that can be performed: passive monitoring and active monitoring. Passive monitoring can be provided by the system directly (*e.g.* a Linux command *top* provides the information of CPU utilisation). Active monitoring requires instrumentation to the applications or operating system to capture specific information (*e.g.* functions or system calls).

2. **analyser**. It analyses data obtained from the monitor. Such as analysing the architecture of systems.

3. **knowledge**. It means knowledge of the system. It acts like a database, which includes the information of system architecture, available actions *et al.*. There are three main approaches [61] used to represent knowledge: *utility concept* (abstract measure of usefulness to a user, *e.g.* the amount of resource available to users, the quality reliability or accuracy of the resource), *reinforcement learning* (construct policies obtained from observing actions) and *bayesian techniques* (a probabilistic technique which is used to provide a way to select from numbers of services or algorithms).

4. **plan**. It takes into account of the monitored data to produce changes and order the produced actions that will be performed on the managed element. Such as ordering the actions and transferring them to executables. A simple case is the event-condition-action (ECA) policy which yields adaptation plans right away from specific events, *e.g.* when 95% of web servers response time exceeds 2 seconds and there are available resources, then increase the number of active web servers.

5. **execute**. It perform changes.

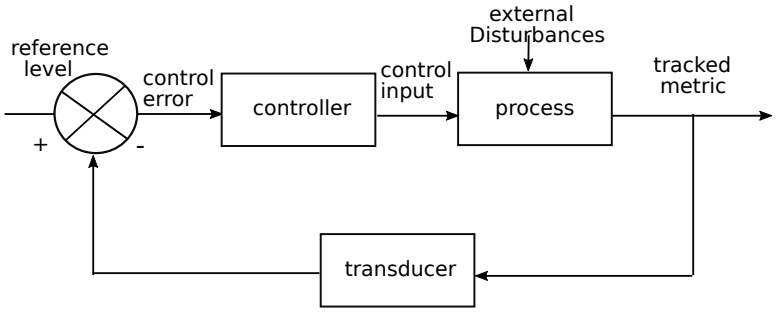


Figure 2.10: The feedback control loop from control theory.

The purpose of the MAPE-K Loop is to develop autonomous control mechanisms to regulate the satisfaction of dynamic requirements, specifically in software systems [69, 62, 6]. An autonomic manager not only implements the phases of the adaptation process, but also provides the interfaces required to sense the environment, and to effect changes on target systems [66]. In the dissertation, feedback control loops are introduced to STM systems to achieve autonomic performance adaptation according to the application runtime behaviour.

2.5.3 Degrees of Autonomicity

IBM [6] defines five levels of autonomicity. (1) On the first level, the automated functions only collects information to support the administrator's decision making. (2) On the second level, the autonomic computing system can act as an advisor to suggest potential actions to be taken by administrators. (3) On the third level, the autonomic computing system can make lower-level decisions and make actions, however, administrators have to frequently make higher-level decisions. (4) On the fourth level, the system still makes lower-level decisions but requires less frequent administrator interventions. (5) On the top level, the systems and components are dynamically managed by rules and polices.

2.5.4 Control Theory in Self-adaptive Systems

Control theory depends on reference control points of system behaviour and corresponding explicit mathematical model specifications [66]. The idea of applying control theory to self-adaptive software system is to find the right mathematical model for modelling the software system behaviour and apply the corresponding treatment.

The theory of autonomic computing overlaps with control theory in the aspect of self-adaptiveness, which is achieved through feedback control loops. A classic feedback control loop from control theory is illustrated in Fig. 2.10. The process as in the figure has a tuning parameter that can be manipulated by a controller. The tracked metric is sensed to obtain the process behaviour and its value is supposed to stay at a reference level. At each instance, the controller computes the difference (called control error) between the value of the tracked metric with the reference level. The controller takes corresponding actions on the process to reduce the control error. However, a nuance [66] between the feedback control loop from control theory and the MAPE-K loop is the complexity of each constituent component and of the overall loop. For example, in the control theory, the control actions are atomic operations for physical actuators with clear causality between inputs and outputs, whereas in the latter, they are a sequence of discrete plans with long lag and uncertain consequences. Albeit the basic principles of the feedbacks in both theories are the same. The details of control theory is out of the scope of the thesis, more information can be found in [70, 71].

2.6 Conclusion Remarks

This chapter has reviewed the related background technologies and techniques on multi-core processors, synchronisation and autonomic computing. It has introduced the topologies of modern processors and discussed the diversity of synchronisation mechanisms. Specifically it has surveyed the significant background on Transactional memory, as transactional memory serves as the synchronisation mechanism used in the dissertation. Two benchmark suites (**EigenBench** and **STAMP**) for TM evaluation are later presented. Lastly, autonomic computing techniques have been covered, which are employed for automatically managing and optimising TM applications and STM platforms at runtime.

Transactional memory, as an alternative parallel programming paradigm, addresses synchronisation issues through transactions. A sequence of instructions are enclosed into a transaction. Transactions execute and access shared objects speculatively without blocking by locks. When conflicts arise, the transaction which detects the conflict can either chooses to abort itself making its previous operations from the current transaction invalid, or the transaction can choose to abort other transactions. The aborted transactions are re-executed immediately or wait for a certain period before resuming. TM can be implemented into software, hardware or hybrid.

Diverse STM systems exist which present their pros and cons. STM systems incorporate many tunable parameters, such as contention manger, version management policy, *etc*. Such tunable parameters are usually set prior to application execution and remain consistent during the entire execution. Little variation of the parameters can be made online to adapt to the diversity of program runtime behaviour. Manually offline tuning the parameters is arduous and less precise. Alongside the complex topologies of multi-core processors and divergence of TM applications, offline tuning becomes more intricate. *Autonomic computing* is able to monitor application and system behaviour at runtime and respond to changes accordingly.

The dissertation focuses on runtime adaptation on thread parallelism and mapping to enhance system performance and describes the corresponding methodologies in the following four chapters. The related work will be given in Chapter 7 in order to illustrate the state-of-art research pertaining to the work of the dissertation and further highlights the contribution of this work.

Chapter 3

Overview of Profile Algorithms and System Architecture

Chapter 3

Overview of Profile Algorithms and System Architecture

STM systems incorporate many tunable parameters such as contention managers, version management policies, *etc*. Such tunable parameters are usually set prior to application execution and remain unchanged during the whole program execution. Little variation of the parameters can be made online by a STM system itself in order to adapt its setting to the diversity of program runtime behaviour. Moreover, with the complex topologies of multi-core processors and divergence of TM applications, more tunable parameters are introduced to a system. Manually offline tuning the parameters is not easy and less precise. On the grounds of runtime interaction between platforms (hardware and software) and applications, application behaviour is more erratic from an offline view. Autonomic computing is able to monitor application and system behaviour at runtime automatically and respond to changes accordingly.

This chapter gives an overview of the proposed system architecture and the methodologies shared in the following three chapters in order to avoid repetition. It serves as an introduction to the work of the thesis on runtime thread parallelism and thread mapping adaptation using autonomic computing techniques. The methodologies on runtime parallelism adaptation is depicted firstly in Chapter 4. Chapter 5 details the runtime adaptation of thread mapping strategies. The coordination of adaptation for parallelism and thread mapping strategies is described in Chapter 6.

This chapter firstly gives the overview of the system architectural organisation (Section 3.1). Then it presents the profiling algorithms in Section 3.2. Lastly, Section 3.3 illustrates the relevant implementation techniques.

3.1 Overview of System Architecture

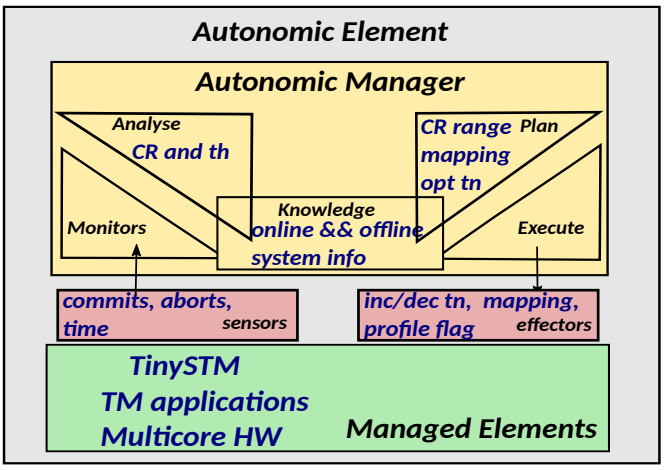


Figure 3.1: Overview of the feedback control loop. *th*, *opt tn* and *mapping* represent throughput, optimum thread number and thread mapping strategy, respectively.

The system architecture is a MAPE-K-shape feedback control loop illustrated in Fig. 3.1, which is an instantiation of the control loop in Fig. 2.9 of Section 2.5. This autonomic element is composed of:

- **Managed elements**. TinySTM (the STM system), TM applications (**EigenBench** and **STAMP**) and the underlying multi-core hardware.
- **Sensors**. The number of commits, the number of aborts and time.
- **Actuators**. Increase or decrease thread number, change thread mapping strategies and set the profile flag. The amount of actuators can vary.
- **Autonomic manager**. This is the controller. When it changes, the corresponding actuators can vary. *E.g.* in Chapter 4, the feedback control loop only manages parallelism degrees, therefore its actuators are shrunk to increment or decrement of parallelism degrees, the profile flag setting; its autonomic manager only determines parallelism degrees.

Under the terminology of control theory, the control objectives of the feedback control loop shared by the following three chapters is to maximise the throughput and diminish the global execution time. This is achieved by adjusting parallelism degree or/and thread mapping strategies at runtime to reduce contention and improve memory resource usage. Recall what is described in Section 2.5.3, IBM defines five levels

of autonomicity. This thesis is concerned with the fourth level of autonomicity on self-optimisation where the autonomic system is able to automatically collect system information and make decisions based on the pre-set rules to optimise application execution time. However, some administrator interventions are still required, such as setting the profiling length.

3.2 Runtime Profiling Approaches

This section covers the general concepts and terminologies which are specifically for the methodologies described in this thesis. Two profiling algorithms are presented afterwards.

3.2.1 General TM Profiling Concepts

Profiling in software engineering refers to a form of dynamic application analysis that measures certain useful events to facilitate application optimisation. Measuring a great amount of events can cause a high time overhead due to the execution of instrumentation code. The instrumented code may impact on application behaviour. Concerning with TM applications, a transaction extends its length by adding instrumentation code that results in a potential change of conflict ratio with other transactions, consequently leading to the change of global contention.

Three events are profiled in the designed architecture, namely the number of commits, the number of aborts and physical time. The number of commits and the number of aborts are addressed as commits and aborts in the rest of the thesis. CR (commit ratio) and throughput (recall that throughput is the commits in a unit of time) are utilised to denote program performance, as they are both sensitive to performance variation of threads. But either by itself is not sufficient enough to represent program performance, as:

- A high throughput shows fast program execution whereas a low throughput represents slow program progress. Nevertheless, a low throughput may be caused by low parallelism or simply just a low number of transactions taking places.

- CR indicates the conflicts among threads. A high CR means low synchronisation time whereas a low CR means a high synchronisation cost. Nonetheless, a low CR can bring a high throughput when a large number of transactions are

executed concurrently, whereas a high CR may give a low throughput due to a small number of transactions executing simultaneously.

Two metrics are often used as timers: physical time and logic time. The logic time is utilised in the thesis, as the length of transaction varies in diverse applications leading to the significant variation of execution time. Commits and aborts are two frequent and meaningful events in TM, are hence good candidates to track the logic events. The number of commits is employed as the timing metric, since it is often a fixed value for an application, while aborts fluctuate significantly with different conflict control strategies and hardware.

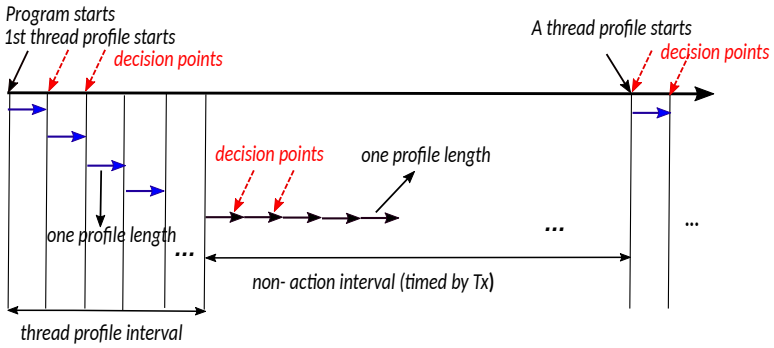


Figure 3.2: The terminologies used for the profiling algorithm in the thesis. One thread interval consists of a continuous sequence of profile lengths within which the parallelism or thread mapping strategy is adjusted, one non-action interval is composed of one or a continuous sequence of profile lengths, within which the thread regulation is suspended.

Ideally, profiling algorithms keep tracing program events and only take actions for optimisation when necessary. The frequency of optimisation actions can significantly impact on application performance. Two profiling algorithms are designed, which are addressed as phase-based profiling and periodical hill-climbing profiling. Fig. 3.2 illustrates how a program is profiled and the terminologies defined in this thesis. A *profile length* is a fixed period timed by commits to gather information, such as commits, aborts and time. A *thread profile interval* is composed of a continuous sequence of profile lengths, within which the parallelism or/and thread mapping strategy is/are adjusted and the CR range is computed. The *non-action interval* consists of one or a continuous sequence of profile lengths, within which the thread regulation is suspended. The duration of thread profile interval and non-action interval are not fixed values as indicated in Fig. 3.2. At the end of one profile length is the *decision point*. The choice of the profile length mainly depends on the total amount of transactions

in an application. The applications with the same magnitude of transactions share the same profile length. For instance, **genome** and **vacation** (two benchmarks from **STAMP** [57]) share the same profile length as the total number of transactions in the two applications are on the same magnitude (*i.e.* 10^6).

3.2.2 Phase-based Profiling Algorithm

A good phase detection method is capable of resolving a phase with a small variance in performance, comparing to the performance variance across the entire applications [72]. A small variance is an indicator demonstrating if the phased detection mechanism detects phase boundaries correctly. Moreover, there exists unstable transition region between two phases, attempting to perform optimisation on the transition region can lead to unpredictable, non-optimal results [72]. A TM application can generate different phases during its execution. Its CR shows distinct difference among each phase, yet fluctuates within a certain range in the same phase.

The phase-based profiling algorithm is useful as the controller only reacts to manipulate the threads when necessary. However it is non-trivial to determine runtime phase boundaries for TM applications. In the proposed methodologies, a CR range is dynamically resolved to indicate the phase. When CR falls out of the CR range, meaning that the program enters a new phase, optimisation actions are invoked (*e.g.* adapting parallelism) accordingly. Two methods are utilised to decide phase diversity.

Simple Phase Detection Algorithm

The first phase detection method is fairly simple. The two thresholds of the CR range are resolved when the optimum parallelism is decided. The two thresholds are the values of the CR generated by the parallelism degrees that are one more or one less than the optimum one.

Advanced Phase Detection Algorithm

The second method is implemented through a feedback control loop that serves as a slave loop to the main control loop. Through continuously modifying the CR range, the actions taken by the main loop responds better to the phase changes. By detecting CR fluctuation in the non-action interval, the main controller decides if the program enters a new phase. When CR remains in a certain range, regulation to the thread is unnecessary as the conflicts in the program has already been minimised by previous control

actions. However, it is onerous to determine such a CR range offline, especially it is unrealistic to set a fixed CR range for the applications with runtime behaviour variation. Moreover, a constant CR range impedes programs to search their optimum parallelism and mapping strategy. Therefore it becomes interesting to dynamically decide a CR range. The derivation of the CR range is based on the optimum CR value. The optimum CR (CR_{opt}) is the value which produced by the optimum parallelism or together with its optimum thread mapping strategy. The values of upper and lower thresholds are the optimum CR plus or minus a factor (δ) of itself, as denoted in Equation 3.1.

$$CR = CR_{opt} \pm \delta * CR_{opt} \tag{3.1}$$

Initially, the value of δ is set to be 10% and is later continuously modified. δ rises 1% when the new predicted parallelism degree equals the previous value or the new value delivers worse performance, meaning that the program still executes in the same phase but the sensors overreact to the CR fluctuation. Therefore it is reasonable to extend the current CR range to reduce the false alert of phase variation. The maximum value of δ is restricted to 0.15.

3.2.3 Periodical Hill-Climbing Profiling Algorithm

The periodical hill-climbing profiling algorithm, as the name indicates, periodically profiles the program and takes control actions. After each thread profile interval, if the configuration stays the same as the previous setting, the non-action interval is doubled otherwise it is reset to the initial value. To prevent the non-action interval from extending too long resulting in slow response to the program behaviour variation, the length of a non-action interval is restricted to a threshold. The initial value of the non-action interval is one profile length. This profiling algorithm is suitable for programs with relatively long and stable phases, although it is less sensitive to the program behaviour change comparing with the previous two phase detection algorithms. Additionally, unnecessary control actions can be taken leading to the program progress under its sub-optimum configurations. This thesis hence only presents performance evaluation based on phase detection algorithms.

3.3 Implementation

This section describes the approaches on how to collect profile information and how to manage threads at runtime.

3.3.1 How to Collect Profile Information

There are two methods for collecting application profile information in a parallel program. A master thread [1] can be employed to record its own information. An alternative way is to collect the information by all threads. The first method requires little synchronisation cost to gather information, but the obtained information may not represent the global view. Additionally, the master thread must not be suspended during the whole program execution, consequently making it terminate earlier than the other threads. This means that fair execution time slots among threads can not be guaranteed. The latter method may suffer from synchronisation cost but the profile information gathered represent the global view. More importantly, a fair execution time strategy can be employed among threads. The second method is utilised in the thesis. The synchronisation cost for gathering information is negligible for most of the TM applications employed.

Events	**TinySTM functions**	**Action performed**
STMInit	stm_init()	initialise STM system
STMExit	stm_exit()	finalise the STM system
ThreadInit	stm_thread_init ()	initialise TM threads
ThreadExit	stm_thread_exit ()	finalise the threads
TxBegin	stm_start()	start a transaction
TxEnd	stm_commit()	commit the current transaction
TxAbort	stm_abort()	abort the current transaction and restarts
TxRead	stm_load()	transaction executes a read operation
TxWrite	stm_store ()	transaction executes a write operation

Table 3.1: The basic STM operations relevant to the experiments.

The profiling mechanism adapted to TM must be generic in order to deal with high diversity of STM systems and TM applications. Such a mechanism should be able to tackle the following two requirements:

[1] A master thread in this thesis means a thread which is randomly selected to perform control actions and collect profile information. It is not necessarily to be the main thread that creates the other threads.

- **Low intrusiveness**. The profiling method should bring low time overhead to applications and should alter little of TM application source code. Two metrics can be utilised to measure the intrusiveness in TM, *i.e.* the execution time and aborts. If execution time and aborts generated by the instrumented applications without performing adaptation vary insignificantly from those generated by original code, a verdict can be reached: application behaviour is not affected.

- **TM system independence**. The profiling method can be adapted to a new STM system easily with little modification and be implemented into a HTM system without significant changes.

The instrumentation code varies when the algorithms are designed for different purposes, therefore profiling intrusiveness may differ. The implementation intrusiveness is illustrated in the following Section (Section 3.3.2).

Although TM systems vary in many aspects as shown in Section 2.3, they rely on some basic operations to perform their functions. Table 3.1 gives the operations which are implemented as functions in STM systems and are relevant to the experiments of the dissertation.

3.3.2 How to Dynamically Control Threads

This dissertation is concerned with two issues, that is dynamic parallelism degree and thread mapping control. To dynamically adjust parallelism, a global monitor is implemented. It enables threads to temporarily give up exclusive access to shared data and later resumes their tasks when some conditions are satisfied. The monitor is a cross-thread lock which consists of the concurrent-access variables by threads. The major variables of the monitor are the *commits*, *aborts*, *throughputs*, two *FIFO* queues recording the suspended and active threads, *current active thread number*, *optimum thread number*, *optimum thread mapping strategy* and *condition variables*. Each thread is associated with one condition variable. A thread waits on its condition variable until a waking signal sent by the other thread. Details about the mechanisms of diverse process synchronisation, such as locks, semaphores and monitors, are beyond the scope of the thesis, more information can be found [73].

A monitor can include several entry points for controlling the threads, where threads are suspended or wakened. The maximum thread number, which is the number of the available cores, is created during application initialisation. Fig. 3.3 illustrates the designed monitor that incorporates three entry points. The first entry point is upon thread

initialisation, where some threads are allowed to pass and the rest are suspended. The second entry point is upon a transaction commit, where commits are accumulated and where the control functions take actions. The third entry point is upon a thread exit, where one suspended thread is wakened by one thread that completes its operations. Fig. 3.3 also demonstrates the interfaces of the sensors, actuators and the controllers. The sensors, which are used to collect commits and aborts reside where the transaction commits and the transaction aborts, respectively. The sensor to measure time reside where the transaction commits and the TM system initialises. The sensor to record time is a Linux kernel function *gettimeofday()*. Fig. 3.4 depicts how the monitor is utilised to control the thread status written in C code. The array *cond_state[]* is utilised to record the condition variables which are booleans. *pthread_cond_wait()* operation sets a condition variable to be false making the corresponding thread suspended. *pthread_cond_signal* sets a condition variation to be true and wakens the corresponding suspended thread. The controller put its control decision into actions by controlling the array *thread_expect_state[]*.

In Fig. 3.3, the control functions, detailed in the following three chapters, are implemented inside the three entry points. The control functions take the inputs collected by the sensors and make corresponding decisions. The monitor acts as an actuator to control parallelism degree. The second actuator for controlling thread mapping strategies is a *pthread* function which affiliates the threads based on the chosen strategy to the corresponding CPU cores. Affiliating a thread to a core can cause thread migration thus performance loss. Recall that, pinning a thread requires reconstruction of cache information if a thread is pinned to a core which resides on another cache level. Therefore it is necessary to avoid frequent change of thread mapping strategies. Thread affinity is managed at two entry points, that is (1) upon thread initialisation where a thread mapping strategy can be specified as an initial strategy, and (2) upon a thread commit where a new thread mapping strategy can be specified and a new wakened thread is assigned to the core where a thread is suspended.

Overhead of the Monitor

A time overhead, which differs on different hardware, is given to each transaction when calling and releasing the monitor. Two factors contribute to this cost: the operation of obtaining and releasing the lock and the time spent to contend for the lock. The latter cost rises significantly when active thread number increases, which gives significant impacts on the applications with short-length transactions (a transaction

```
/*adjust functionalities*/
control_func(time,commits,aborts){
 ...
 adjust tn;
 adjust thread mapping strategy;
 decide control decision frequency;
 ...
}
```

```
//The entry point
stm_thread_init(){
 ...
 control_func(time,commits,aborts);
 ...
}
```

```
//The entry point.
/*The info of all threads are synchronised at this entry point*/
stm_commit() {
  ...
 commit sensor;//collect commits
 time sensor; //record time
 ...
 control_func(time,commits,aborts);
 ...
}
```

```
stm_abort(){
 ...
 abort sensor;//collect aborts;
 ...
}
```

```
//The entry point
stm_thread_exit(){
 ...
 control_func(time,commits,aborts);
 ...
}
```

```
stm_init(){
 ...
 time sensor;
 ...
}
```

Figure 3.3: The three entry points of the monitor and the control functions. The *monitor* in the figure is a cross-thread lock which includes the concurrent access variables by threads. The code is written in pseudo C. tn stands for active thread number.

incorporating a small number of operations). Table 3.2 and Table 3.3 illustrate performance difference with and without the monitor when applications executing with the static optimum parallelism. The results are generated by the monitor when called every commit. On the UMA platform (used in the thesis), the overhead caused by calling locks is negligible for the transaction with medium length and long length, more

```
/*suspend threads*/
pthread_mutex_lock(&(monitor_t->monitor));
if ((mod_monitor->thread_expect_state[thread_id])==false)
    while ((monitor_t->thread_expect_state[thread_id])==false)
              monitor_t->active_tn--; //obtain the number of threads suspended
              pthread_cond_wait(&(monitor_t->cond_state[thread_id]),&(monitor_t->monitor));
              monitor_t->active_tn++;
pthread_mutex_unlock(&(monitor_t->monitor));
```

```
/*waken threads*/
pthread_mutex_lock(&(monitor_t->monitor));
pthread_cond_signal(&(monitor_t->cond_state[thread_id]));
pthread_mutex_unlock(&(monitor_t->monitor));
```

Figure 3.4: A snapshot of C code on suspending and wakening threads. This is one of the control functions in Fig. 3.3

specifically is less than 2%. On the NUMA machine, the majority applications indicate around 4% time overhead with the monitor. Time fluctuations are shown on **genome** (around 4% on UMA), as its execution time is short. **intruder** and **ssca2** demonstrate 20.6% and 224.6% time overhead with the monitor on the UMA due to its short-length transactions. On the NUMA platform, the overhead cause by the monitor for the two applications is smaller than that on UMA, *i.e.* around 5% and 12% for **intruder** and **ssca2**, respectively. It is not surprising to observe such a significant overhead difference for **ssca2** between the two machines, since the optimum thread number executed on NUMA is 6 while 24 on UMA. The time spent in contending for the lock is hence higher on UMA in this case.

The overhead caused by calling the monitor can be reduced through diminishing its calling frequency. More specifically, the monitor is called every 100 commits rather than every commit. As a time stall is added at commit time, it makes the contention manger act similar as a *backoff* policy (except the *backoff* policy gives proportional time stall based on the retry times). Therefore some applications when the global monitor is used show slightly better or worse performance than the ones without the monitor.

Deadlock Avoidance

Some applications, such as **genome** and **ssca2**, incorporate multiple thread barriers which conflict with the monitor, thus deadlock. To tackle the issue, an additional function is inserted before a thread barrier to avoid the deadlock. Once a thread encounters a barrier, it wakens one suspended thread. When all the threads have passed the barrier, the last thread sets parallelism back to the optimal value. Thread control is disabled

Application	tn	without monitor		with monitor	
		time (s)	aborts	time (s)	aborts
EigenBench (one phase)	12	31.8	1422425	31.6	1427577
EigenBench (three phases)	12	57.3	2406000	56.7	2395137
EigenBench (two phases)	2	36.3	272180	35.8	247519
intruder	6	10.7	61687402	12.9	44105349
ssca2	24	6.1	18333	19.8	19428
genome	4	4.6	9706425	4.8	7743698
vacation	8	9.4	233214	9.5	208642
yada	8	9.1	80555689	9.2	78392329
labyrinth	20	37.1	217	37.3	219

Table 3.2: Intrusiveness of the global monitor for applications on the UMA platform. *tn* represents the optimum parallelism degree. The monitor is called every commit.

Application	tn	without monitor		with monitor	
		time (s)	aborts	time (s)	aborts
EigenBench (one phase)	12	33.9	1438002	33.9	1435445
EigenBench (three phases)	12	59.4	2418888	59.8	2418097
EigenBench (two phases)	4	28.4	2237076	28.2	2232561
intruder	6	9.3	36101578	9.8	28621899
ssca2	6	13.2	708	14.8	238
genome	4	4.0	3614305	4.0	3211683
vacation	8	7.3	168524	7.2	163164
yada	12	7.9	125357974	8.0	119209845
labyrinth	32	25.5	326	24.2	312

Table 3.3: The intrusiveness of the global monitor for applications on the NUMA platform. *tn* represents the optimum parallelism degree. The monitor is called every commit.

when the first thread meets a barrier until the last thread passes the barrier, as the application behaviour is unstable at this period.

Round-robin Thread Rotation

To avoid thread starvation, round-robin thread rotation is employed to periodically awaken early suspended threads and suspend the running threads having executed longest time. There are two ways to implement thread round-robin rotation algorithm: timestamp based round-robin and First-In-First-Out (FIFO) queue. The timestamp algorithm marks an explicit time line for each thread (the time when it is wakened and the time when it is suspended), however, it requires to compare the timestamp of all

the threads for each rotation operation. The FIFO queue algorithm does not include the explicit time line, however, it performs simple push-in and pop-out operations to achieve the thread rotation. The latter algorithm is chosen in this thesis. Two FIFO queues are implemented, with one recording the suspended threads and one recording the active threads. Thread starvation avoidance is necessary for the application with a set amount of tasks allocated to each thread. This is the case for **EigenBench**, without thread rotation the suspended threads starve, which not only slows down application execution, but also changes application runtime behaviour. However in some applications such as **yada** from **STAMP**, the tasks are allocated to each thread at runtime, and no tasks are assigned to the suspended threads, therefore applications likewise require no round-robin thread rotation. Table 3.4 lists the effect on fair execution time among threads by round-robin thread rotation for diverse applications. More details of the benchmark analysis will be explained together with performance evaluation in the following chapters.

Application	**Effect**	**Applications**	**Effect**
EigenBench (one phase)	yes	**intruder**	no
EigenBench (three phases)	yes	**ssca2**	no
EigenBench (two phases)	yes	**genome**	no
vacation	yes	**labyrinth**	no
yada	no		

Table 3.4: The effect of round-robin thread rotation on applications.

3.4 Conclusion Remarks

This chapter gives an overview of the system architecture, presents the methods on application phase detection and illustrates the implementation details. It provides the techniques which are shared by the next three chapters. The system architecture is described as a feedback control loop in MAPE-K shape which varies in sensors, actuators and autonomic manager in following three contribution chapters. Autonomic computing employs feedback control loops, as it systematically depicts the design objectives and system architecture. In addition, it develops autonomous control mechanisms to satisfy runtime requirements. However, this chapter offers little insights into the pros and cons of different phase detection approaches, as such will be better demonstrated with performance evaluation. The implementation part describes the design details and

discusses its advantages as well as limitations.

The next chapter presents the dynamic parallelism adaptation approaches. It describes two parallelism adaptation approaches and illustrates their efficiency through performance evaluation.

Chapter 4

Autonomic Parallelism Adaptation

Chapter 4

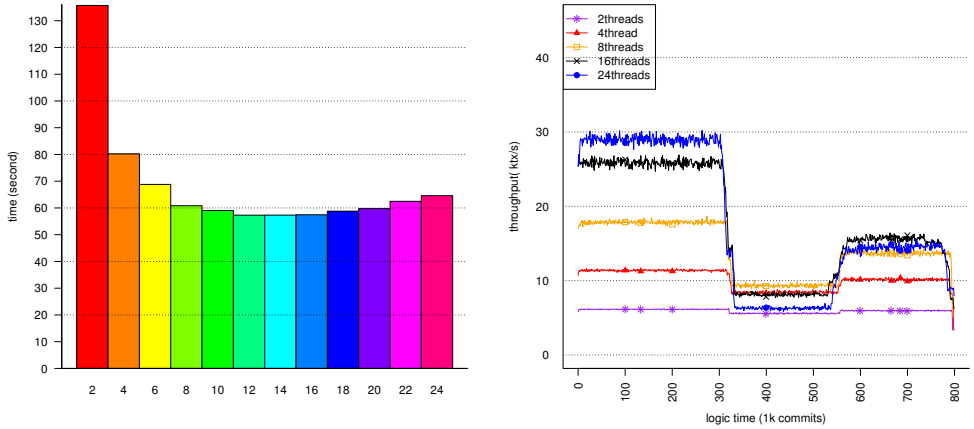
Autonomic Parallelism Adaptation

4.1 Introduction

The parallelism degree can significantly impact on TM application performance. *E.g.* Fig. 4.1(a) illustrates execution time difference of static parallelism degrees for one application from **EigenBench**. A better choice of parallelism degree can boost application performance. However, it is onerous to decide an optimum parallelism degree offline especially for the program with online behaviour variation. Apropos of a program with online behaviour fluctuations, no unique parallelism degree can yield optimum performance. For instance, Fig. 4.1(b) indicates throughput fluctuations of **EigenBench** for certain parallelism degrees. This application exhibits three phases where no single parallelism degree can always achieve the highest throughput at each phase. Additionally, the optimum parallelism degree varies when the same application runs on a different platform. Therefore, the natural solution is to audit a program at runtime and alter its parallelism degree when necessary.

This chapter presents the algorithms on parallelism adaptation for STM systems using control techniques and technologies. Section 4.2 and Section 4.3 describe two autonomic models that detect near-optimum parallelism at runtime. The parallelism adaptation algorithms are described through control theory view. Benchmark settings are presented later in Section 4.4 to clarify characteristics of the applications. Following that, Section 4.5 firstly illustrates application performance diversity when different static parallelism[1] is applied. It then demonstrates the performance evaluation on dynamic parallelism adaptation.

[1] In this thesis, static parallelism refers to the parallelism degree which is selected offline and running during the whole application execution without adaptation.



(a) time difference (b) throughput comparison

Figure 4.1: An illustration of necessity of dynamic parallelism adaptation for **Eigen-Bench**.

4.2 Simple Model for Parallelism Adaptation

This section presents a model (addressed as *simple model*) that dynamically searches the near-optimum parallelism degree for each phase of an application.

4.2.1 Overview of the Profiling Algorithm

The profiling procedure is depicted in Fig. 4.2. The parallelism profiling procedure starts once the program starts. Initially, the program initialises the thread number that equals the core number, but only 2 threads are active and the rest are suspended. At each decision point, which corresponds to one state of the automaton in Fig. 4.3(b), the controller is activated to increase or decrease the parallelism degree continuously or suspend the parallelism regulation. The above procedure continues until the program terminates.

4.2.2 Feedback Control Loop of the Simple Model

Fig. 4.3(a) gives the structure of the complete platform which forms a MAPE-K (definition in Section 2.5.2) feedback control loop. The structure of the autonomic manager is shown in Fig. 4.3(b). The autonomic manager, which can be also seen as the controller, is described as an automaton in this thesis. The automaton is composed of four states. One state is called at each decision point.

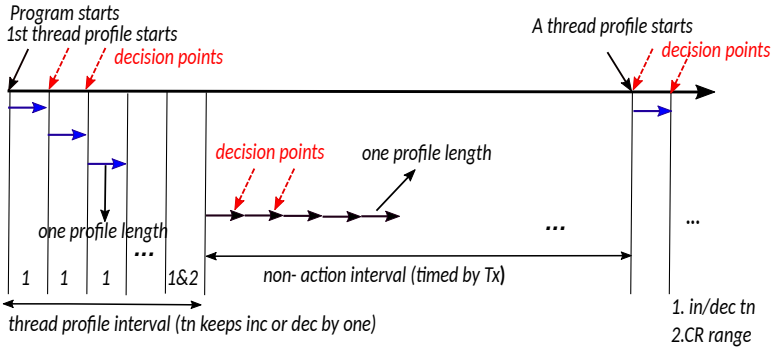
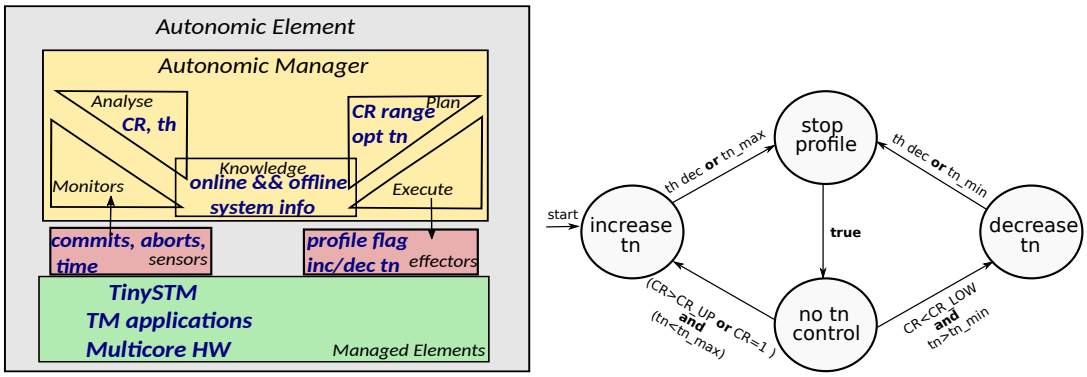


Figure 4.2: Profiling procedure for the simple model. The control actions are taken at each decision point (marked by dashed red arrow).



(a) The MAPEK-shape feedback control loop.

(b) The structure of the autonomic manager

Figure 4.3: The feedback control loop of the simple model.

Inputs and Outputs

The inputs of the loop are commits, aborts and physical time (see Section 2.3.1 and Section 3.2.1 for definition). The actions are to increase or decrease parallelism degrees and setting the profile flag.

Three Decision Functions

The control loop is activated at each decision point. Three decision functions cooperate to make decisions: a *parallelism decision function*, a *profile decision function* and a *CR range decision function*. Each decision point in Fig. 4.2 corresponds to one state of the automaton in Fig. 4.3(b). The corresponding decision functions are called to make decisions at each state. The *parallelism decision function* and the *profile decision function* are detailed in the following paragraph. The *CR range decision function* that has been described in Section 3.2.2 is employed to detect program phases.

The automaton begins with the state ***increase tn***, since the thread number is set to the minimum at the starting point. Starting from the minimum thread number can avoid the excessive conflicts among threads which hinders program progress. In each thread profile interval, the parallelism can either continuously increase or decrease. The direction of parallelism regulation (increase or decrease) is determined by the *profile decision function*. If the current throughput is greater than the previous throughput, one thread is wakened or suspended and the current throughput is recorded as the maximum throughput. The state shifts to ***stop profile*** when the current throughput is less than the maximum throughput. At the final decision point of a thread profile interval (at the state ***stop profile***), the parallelism is set to the value that yields the maximum throughput. A new CR range may be computed at the end of a thread profile interval as detailed later in this section. Then the automaton shifts from the state ***stop profile*** to the state ***no tn control*** that corresponds to non-action interval in Fig. 4.2. At each decision point of a non-action interval, the *profile decision function* decides if a new parallelism profile interval is needed. More specifically, if the CR falls into the CR range, the program stays in the ***no tn control*** state. Otherwise a boolean value is set indicating the direction of the parallelism regulation. Specifically, the automaton shifts to ***increase tn*** if CR is higher than the upper CR threshold or ***decrease tn*** otherwise. It is worth noting that, in case the maximum value of the upper threshold is 100%, and the program CR is 100% (when only reads operation in the transactions or no conflicts among transactions), a higher parallelism degree to the program is assigned.

The throughput often fluctuates before reaching the optimum value as shown in Fig. 4.4. To prevent a parallelism profiling procedure from terminating at a local maximum throughput, the parallelism profiling procedure continues until the throughput decreases over 10% of the maximum value (10% is an empirical value that is tunable).

The *parallelism decision function* described in the above paragraph is called in the state ***increase tn*** and ***decrease tn***. The runtime phases are indicated by CR fluctuations. There is a natural fluctuation of CR at runtime within the same phase. Ideally, a control action should only take place when a new phase begins. It is onerous to determine a CR range offline, especially it is impossible to set a fixed CR range for some programs with online performance variation. Also, a constant CR range impedes programs to search their optimum parallelism. Therefore it becomes necessary to dynamically resolve a CR range. At the end of a thread profile interval, a new CR range is prescribed. The function is activated at the state ***stop profile*** corresponding to the last decision point of a thread profile interval. The simple thread adaptation model utilises the simple

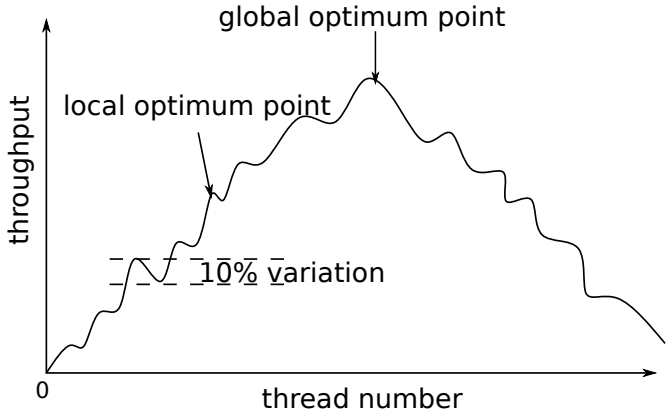


Figure 4.4: Throughput fluctuation. The throughput fluctuates before reaching the maximum point.

phase-based profiling algorithm as its *CR range decision function*, since this decision function does not give additional cost to the model. Recall that, the simple phase-based profiling algorithm decides its CR range by recording the CR generated by one more and one less parallelism degree than the optimum value (see Section 3.2.2 for details).

4.3 Probabilistic Model for Parallelism Adaptation

The approach to alter the parallelism degree by one at each decision point occupies long profiling time which slows down execution. This section presents a probabilistic model that predicts the parallelism based on the information of one profile length. Likewise the profiling procedure stated in Section 4.2, Fig. 4.5 depicts the profiling procedure for probabilistic model. As illustrated in Fig. 4.5, the probabilistic model requires two profile lengths to obtain the near-optimum parallelism and possibly two additional profile lengths to determine the CR range. In contrast, the simple model employs at least two profile lengths each time to achieve this goal. Since the feedback control loop is similar to that of the simple model, with an exception in the autonomic manager, the rest of this section only describes the design of the autonomic manager.

4.3.1 The Autonomic Manager

As illustrated in Fig. 4.6, the automaton commences from the ***predict tn*** state which predicts a near-optimum parallelism degree. The predicted parallelism is applied for one subsequent profile length. Following that the automaton unconditionally shifts to

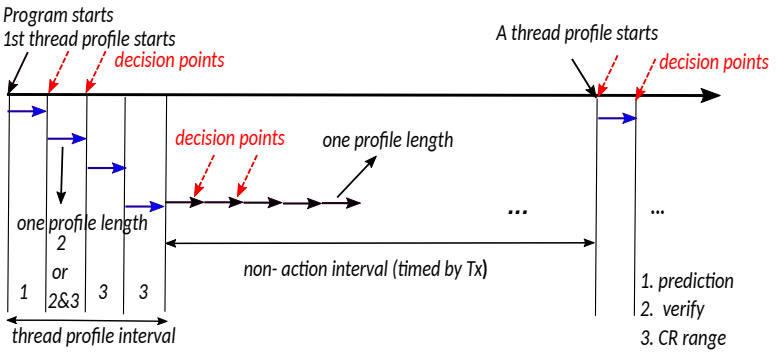


Figure 4.5: Profiling procedure for the probabilistic model. The control actions are taken at each decision point (marked by dashed red arrow).

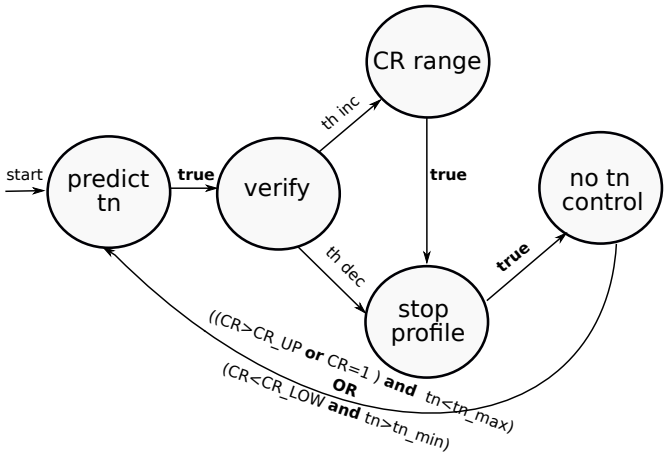


Figure 4.6: The controller of the probabilistic model described as an automaton. th stands for throughput and tn means the thread number.

the *verify* state verifying the correctness of the predicted parallelism. The new predicted parallelism is only kept subsequently when the current throughput is larger than the previous parallelism. This leads to the state shifting to the ***CR range*** state where a new CR range is prescribed. Otherwise it recovers the previous value (CR range remains unchanged in this case). The ***stop profile*** state disables the parallelism profile action when the parallelism does not alter after verification or when a new CR range is resolved. Contrary to the simple model, the probabilistic model requires an individual state to obtain the CR range. The simple model continuously increases or decreases the parallelism degree until reaching the optimum throughput, therefore it requires no additional state for CR range decision.

The following section only describes the *parallelism decision function* for the probabilistic model, as the *profile decision function* and *CR range decision function* performs the same as those in the simple model.

4.3.2 Parallelism Prediction Decision Function

This section describes a probabilistic model which serves as the *parallelism decision function*. This probabilistic model yet has its limitations, as it is based on two assumptions:

1. assuming that the same amount of transactions are executed in the active threads during a fixed period, as every thread shows similar behaviour in the TM applications used in this thesis;

2. assuming that the probability of one commit (a transaction successfully accomplishes its operations encountering no conflicts with the remaining transactions) approaches a constant, as there is enough amount of transactions executed during the fixed period, making the probability of a conflict between two transactions approaches a constant.

Within a fixed period L_0 during an application execution, assuming that the average length of transactions (including the aborted transactions and the committed transactions) is L , thus the number of transactions N executed during L_0 can be expressed in equation (4.1).

$$N = \frac{L_0}{L} \cdot n = \alpha \cdot n \tag{4.1}$$

Where n stands for the number of active threads during L_0, N contains both aborts and commits, and $\alpha = \frac{L_0}{L}$.

We assume that the probability of the conflicts p between two transactions is independent from the current active parallelism degree, thus independent from the number of active transactions. Therefore during the L_0 period, one transaction can commit if it encounters no conflicts with other active transactions. The probability of a commit can be expressed in Equation (4.2).

$$P(X_i = 1) = (1-p)^{(N-1)} = q^{(N-1)} \tag{4.2}$$

Where $q = 1 - p$, which stands for the probability of a commit between two transactions. However, the transactions executed in a sequence within the same thread do not

cause conflicts among each other, the probability of a commit in Equation (4.2) is hence lower than the reality. Presumably during L_0 period, each thread approximately execute the same number of transactions $\frac{N}{n}$. Therefore the number of transactions causing conflict are reduced to $N - \frac{N}{n}$. So Equation (4.2) can be modified as in Equation (4.3).

$$P(X_i = 1) = q^{(N-\frac{N}{n})} = q^{\alpha(n-1)} \tag{4.3}$$

Equation 4.3 is correct only if there is a large amount of transactions executed during L_0 period making the probability of conflicts p between two transactions approaches a constant, thus q approaches a constant.

Under the terminology of probability theory, X_i is a random variable with $X_i = 1$ if the transaction i is committed, $X_i = 0$ if aborted. X_i follows a Bernoulli law of parameter $q^{(N-\frac{N}{n})}$.

Let T represent the throughput. In a unit of time, the throughput can also be expressed as $T = \sum X_i$ and CR can be expressed as $CR = \frac{T}{N}$, as T is a random variable which follows a binomial distribution $B(N, q^{(N-\frac{N}{n})})$. The expected value of T is therefore to be:

$$E[T] = N.q^{(N-\frac{N}{n})} = \alpha n q^{\alpha(n-1)} \tag{4.4}$$

Hence the expected value of CR is:

$$E[CR] = \frac{E[T]}{N} = q^{\alpha(n-1)} \tag{4.5}$$

Equation (4.4) can be rewritten as a function from n to T as shown in Equation (4.6).

$$T(n) = \alpha.n.q^{\alpha(n-1)} \tag{4.6}$$

To obtain the value of n where the throughput could reach the maximum, the derivation of Equation (4.6) is computed as shown in Equation (4.7).

$$T'(n) = \alpha q^{\alpha(n-1)} + \alpha^2 n q^{\alpha(n-1)} \ln(q) \tag{4.7}$$

Therefore

$$\begin{aligned} T'(n_{opt}) = 0 &\Leftrightarrow \alpha q^{\alpha(n_{opt}-1)} + \alpha^2 n_{opt} q^{\alpha(n_{opt}-1)} \ln(q) = 0 \\ &\Leftrightarrow q^{\alpha(n_{opt}-1)}.(\alpha + \alpha^2 n_{opt} \ln(q)) = 0 \\ &\Leftrightarrow \alpha + \alpha^2 n_{opt} \ln(q) = 0 \\ &\Leftrightarrow n_{opt} = -\frac{1}{\alpha \ln(q)} \end{aligned} \tag{4.8}$$

Where n_{opt} stands for the optimum value of n which is the optimum parallelism degree.

From Equation (4.5), one can derive $q = CR^{\frac{1}{\alpha(n-1)}}$. Then Equation (4.8) can be rewritten as follows:

$$n_{opt} = -\frac{n-1}{\ln(CR)} \tag{4.9}$$

Where n_{opt} stands for optimum thread number, n stands for the number of current active threads and CR is the current commit ratio.

4.4 Benchmark Setting

This chapter as well as the two following chapters presents performance evaluation on six different **STAMP** [57] benchmarks and three applications from **EigenBench** [56]. **EigenBench** and **STAMP** are widely used for performance evaluation on TM systems. The data sets of the selected applications cover a wide range of parameters from short-length to long-length transactions, from short to long program execution time, from low to high program contention. Table 4.1 presents the qualitative summary of each application's runtime transactional characteristics (based on the statistics from the UMA platform): Tx (transaction) length or Tx size (the number of instructions per transaction), execution time, and contention (the global contention). The classification is based on the application executed with its static optimum parallelism on the UMA machine. A transaction with execution time between 10 *us* and 1000 *us* is classified as medium-length. The contention between 30% and 60% is classified as medium. The execution time between 10 seconds and 30 seconds is classified as medium. It is worth noting that an application with short or very short length transactions requires to reduce the frequency of calls to the monitor. Recall that, it is introduced in Chapter 3, the monitor is utilised to dynamically control parallelism degree and collect profile information.

Three applications from **EigenBench** are evaluated, *i.e.* one application with stable behaviour (one phase), one with two phases, one with three phases. They are selected and configured in this thesis to serve as complementary benchmarks to **STAMP** for performance evaluation on special issues. **EigenBench**, with one phase, presents very stable runtime behaviour, hence it is ideal to demonstrate the overhead of control actions. Fig. 4.7(a) provides the inputs for an application with stable behaviour. The application, with two phases, is designed to verify if a change of parallelism requires a change of thread mapping strategies. As most of the transactions in each **STAMP**

Application	Tx length	Execution time	Contention
EigenBench	medium	long	medium
ssca2	very short	short	low
intruder	short	medium	high
genome	medium	short	high
vacation	medium	medium	low
yada	medium	medium	high
labyrinth	long	long	low

Table 4.1: Qualitative summary of each application's runtime transactional characteristics. The classification is based on the application with its optimum parallelism applied on the UMA machine.

application usually have very similar behaviour [5] making them unsuitable for evaluation of the dynamic thread mapping approaches. The application with three phases is utilised to demonstrate the necessity of runtime parallelism adaptation. This thesis utilises two approaches to enable dynamic phases for **EigenBench**. The first method is to modify the source code of **EigenBench**, since the original source code presents no runtime phase variation. **EigenBench** includes three different arrays which provide the shared transactional access (Array1), private transactional access (Array2) and non-transactional access (Array3). Dynamically varying the size of Array1 makes the conflict rate vary. More specifically, within the first 40% of total number of the transactions, the size of Array1 keeps the value given by the input file. From 40% to 70%, the array size is shrunk to 16% of the original value and afterwards the size is set to 33% of the given value. This data set is shown in Fig. 4.7(b). A second approach to create several phases is to provide several data sets and execute them in a sequence, as each data set can give individual behaviour. However, the execution order of the data sets is randomised [56] in **EigenBench**. In order to ensure consistent behaviour at each execution, the source code has been slightly modified to disable randomisation. Fig. 4.7(c) illustrates the data sets which provide two distinguishing phases.

Six different applications from **STAMP** are presented, namely **ssca2**, **intruder**, **genome**, **vacation**, **yada** and **labyrinth**. Two applications namely **bayes** and **kmeans** from **STAMP** are not taken into account in the thesis. **bayes** exhibits non-determinism [58]: the ordering of commits among threads at the beginning of an execution can drastically affect execution time. The number of commits shows significant differences during each execution for **kmeans**, therefore, it is excluded from performance evaluation. The inputs of the six selected applications are detailed in Fig. 4.8.

loops	16667
A1	35536
A2	1048576
A3	8192
R1	30
W1	30
R2	20
W2	200
R3i	10
W3i	30
R3o	10
W3o	10
NOPi	0
NOPo	0
Ki	1
Ko	1
LCT	0

(a) one phase

loops	33333
A1	145530
A2	1048576
A3	8192
R1	30
W1	30
R2	20
W2	200
R3i	10
W3i	30
R3o	10
W3o	10
NOPi	0
NOPo	0
Ki	1
Ko	1
LCT	0

(b) three phases

		*R1	0	35
		*W1	0	45
loops	3333	*R2	0	200
A1	95536	*W2	0	100
A2	1048576	*R3o	0	10
A3	819200	*W3o	0	10
NOPi	0	*R1	1	300
NOPo	0	*W1	1	220
Ki	1	*R2	1	100
Ko	1	*W2	1	50
LCT	0	*R3i	1	0
M	2	*R3o	1	0
		*W3o	1	0

(c) two phases

Figure 4.7: Inputs of EigenBench applications for 24 threads.

ssca2	-s20 -i1.0 -u1.0 -l3 -p3
intruder	-a8 -l176 -n109187
genome	-s32 -g32768 -n8388608
vacation	-n4 -q60 -u90 -r1048576 -t4194304
yada	-a15 -i inputs/ttimeu1000000.2
labyrinth	-i random-x1024-y1024-z7-n512.txt

Figure 4.8: The inputs of **STAMP**

4.5 Performance Evaluation

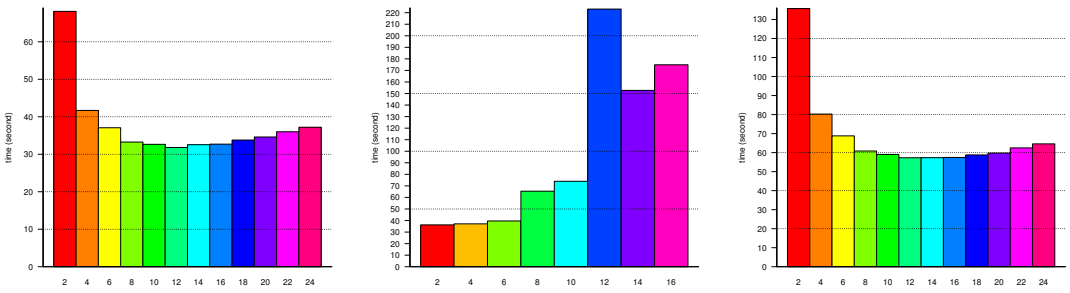
This section presents the performance evaluation of the two dynamic models on two hardware platforms (the UMA and NUMA machines). The UMA machine has uniform main memory access from each core. In contrast, the NUMA machine which possesses distributed memory, has a non-uniform memory access, meaning that the access time from a core to its local memory is faster than that to remote memory. In the performance evaluation, the node interleaving for the NUMA machine is disabled. The NUMA machine behaves similarly as a UMA machine when its node interleaving is enabled.

The results generated by the two autonomic models are compared against the results of static parallelism which presents the best, average and worst performance.

Such comparison is utilised to demonstrate that the autonomic models can outperform static parallelism even if an unknown application is given. First of all, this section manifests the performance of static parallelism. Secondly, it presents the results of the execution time comparison between two autonomic models and the static parallelism. Following that, it illustrates the runtime parallelism variation adjusted by the two models. Lastly, the throughput comparison is given to illustrate the efficiency and correctness of the autonomic models. The predicted parallelism is correct if the throughputs generated by the autonomic models converges to or rival the static parallelism which generates the highest throughput. The maximum parallelism degrees are 24 (UMA) or 32 (NUMA) respectively, which is the number of the available cores of the hardware. The minimum parallelism degree is two, as only parallel applications are of concern in this thesis. The simple model starts with two threads to avoid excessive contention, on the grounds that the model only regulates one thread number at each decision point. *Per contra*, the probabilistic model starts with 24 threads which equals the maximum core number of the UMA platform as the parallelism prediction of the probabilistic model relies on execution of a large amount of transactions. All the applications are executed 10 times and results are the average execution time.

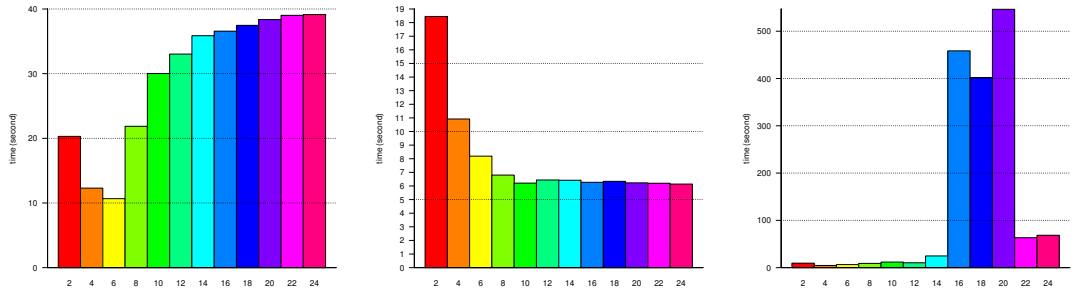
4.5.1 Performance of Static Parallelism

Fig. 4.9 and Fig. 4.10 illustrate the performance differences on diverse static parallelism on the UMA machine. Fig. 4.11 and Fig. 4.12 depict performance on the NUMA machine. It is worth noting that the execution time of **EigenBench** with two phases increases drastically, therefore Fig. 4.9(b) and Fig. 4.11(c) only present the execution time up to 16 parallelism degree.

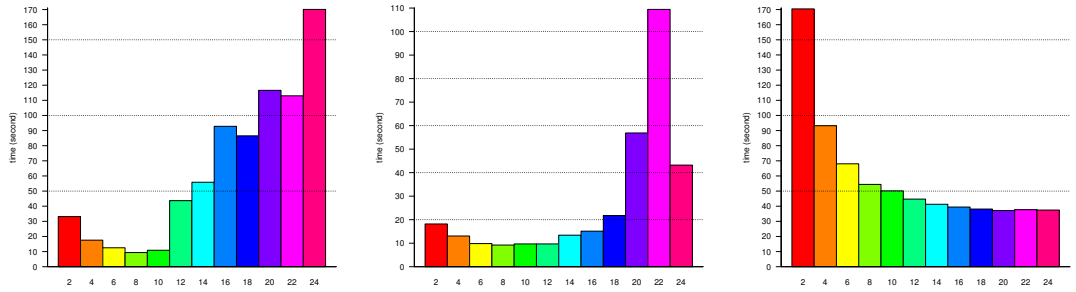


(a) one phase (b) two phases (c) three phases

Figure 4.9: Time comparison of **EigenBench** for static parallelism on UMA.

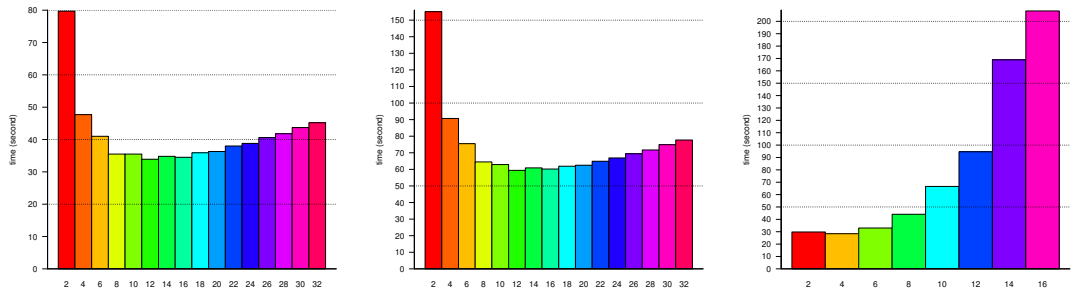


(a) **intruder** (b) **ssca2** (c) **genome**



(d) **vacation** (e) **yada** (f) **labyrinth**

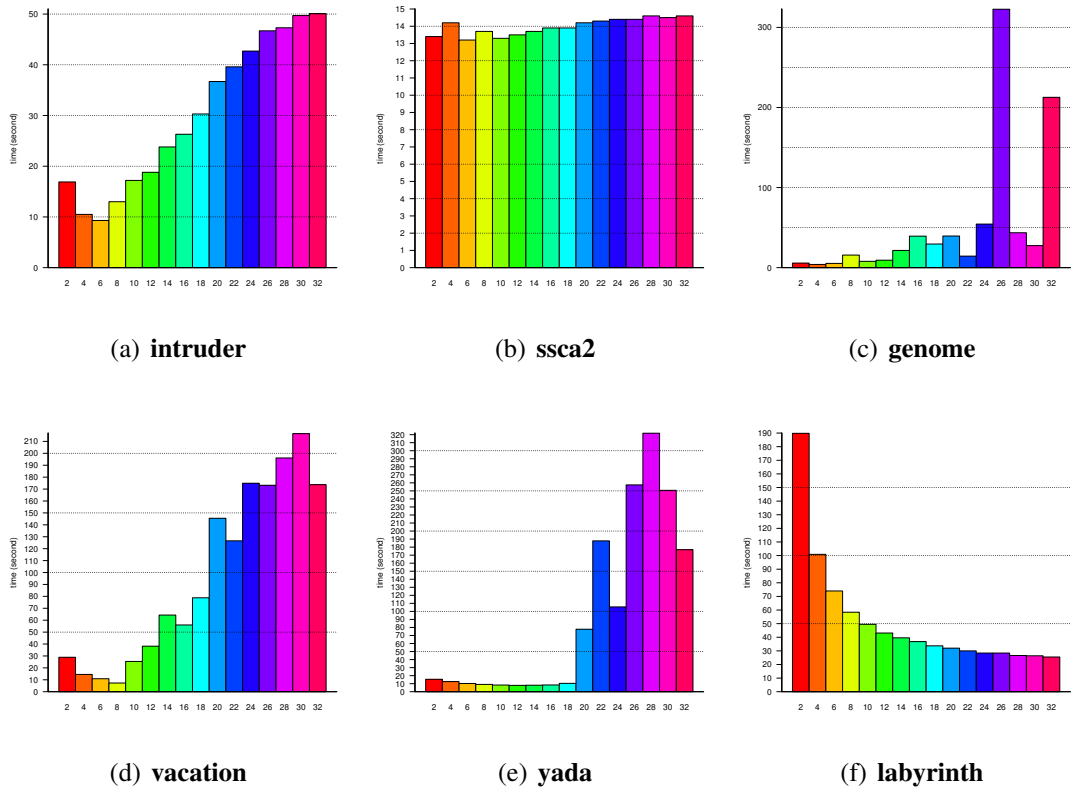
Figure 4.10: Time comparison for **STAMP** for static parallelism on UMA.



(a) one phase (b) two phases (c) three phases

Figure 4.11: Time comparison of **EigenBench** for static parallelism on NUMA.

Some applications demonstrate unstable behaviour with the ascent of its parallelism degree, such as **EigenBench** (two phases), **genome**, **yada**, this is due to the TM mechanisms on transaction conflicts detection and resolution. As indicated in the figures of the time comparison for static parallelism, performance of an application



(a) **intruder** (b) **ssca2** (c) **genome**

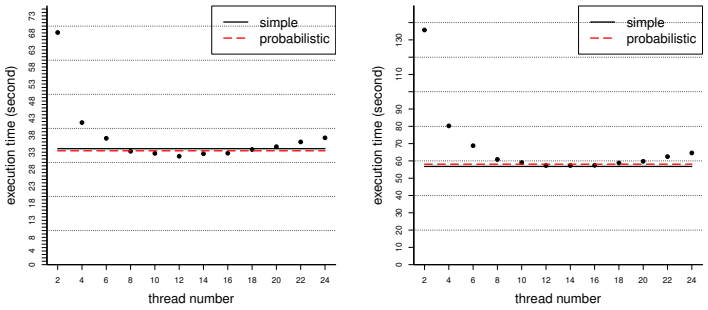
(d) **vacation** (e) **yada** (f) **labyrinth**

Figure 4.12: Time comparison for **STAMP** for static parallelism on NUMA.

can depend on its parallelism degree. Its optimum parallelism degree can also differ when the same application runs on a different platform (*e.g.* **yada**). Therefore adjusting parallelism degree can not only improve application performance, but also ensures stability. Unstable behaviour is often produced by the parallelism degree which causes high program contention due to cascading transaction aborts. Nevertheless, **ssca2** displays little performance difference among static parallelism degrees, regardless of the hardware it runs. This benchmark, therefore, will hardly benefit from dynamic parallelism adaptation.

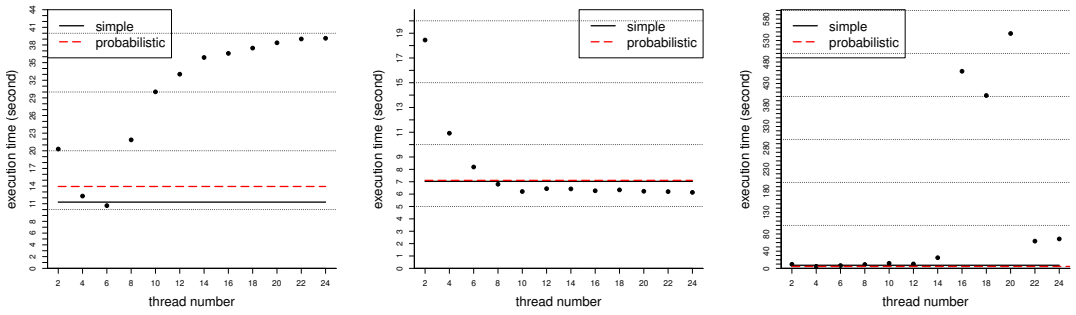
4.5.2 Performance Evaluation on the UMA Platform

Fig. 4.13 and Fig. 4.14 illustrate the execution time comparison with diverse static parallelism and adaptive parallelism for **EigenBench** and **STAMP**. The dots represent the execution time with different static parallelism. The solid black line stands for execution time with the simple model and the dashed red line gives the execution time with the probabilistic model.

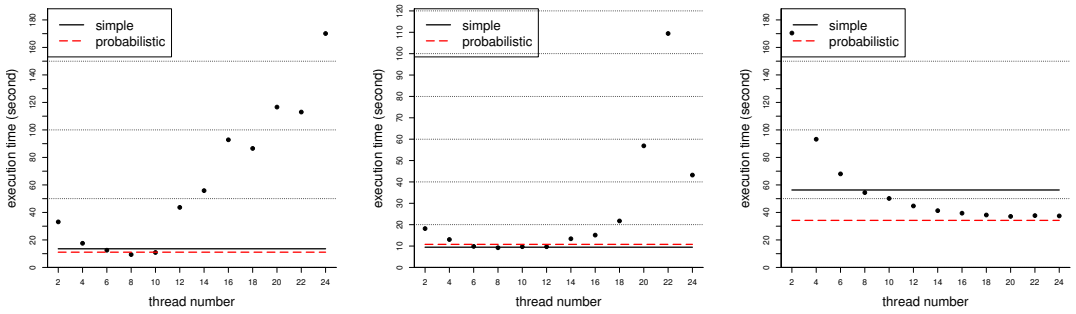


(a) one phase (b) three phases

Figure 4.13: Time comparison of static and adaptive parallelisms for **EigenBench** on UMA. The dots represent the execution time with static parallelism.



(a) **intruder** (b) **ssca2** (c) **genome**



(d) **vacation** (e) **yada** (f) **labyrinth**

Figure 4.14: Time comparison of static and adaptive parallelism for **STAMP** on UMA. The dots represent the execution time with static parallelism.

According to Fig. 4.13 and Fig. 4.14, the adaptive parallelism outperforms the majority of the static parallelism. The probabilistic model shows better performance on applications, *i.e.* **genome**, **vacation**, **labyrinth** against the simple model, yet it

indicates performance degradation on **yada** and **intruder** against the simple model. Both models present similar performance on **EigenBench** and **ssca2**. Table 4.2 and Table 4.3 detail the performance comparison. The digits in the brackets are the static parallelism degrees which give the best and the worst performance, respectively. The symbol "plus" (+) means performance gain against the compared value.

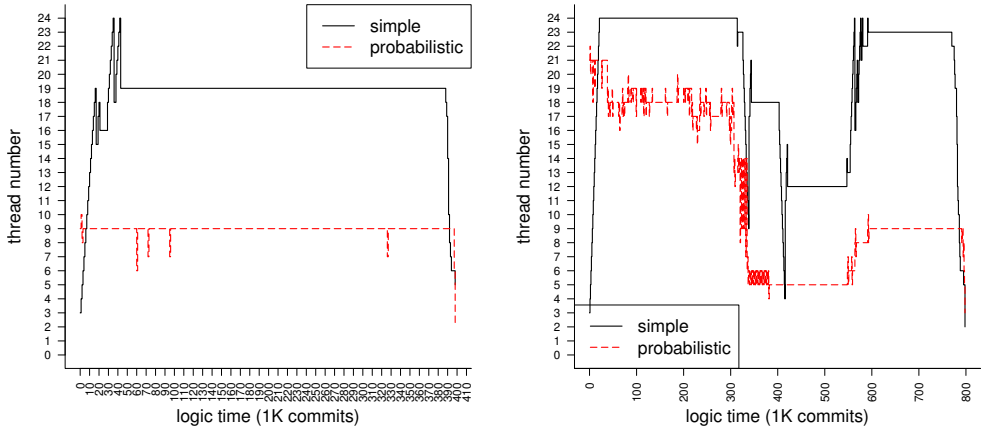
benchmarks	best case	average	worse case
EigenBench (one phase)	-7% (12)	+10%	+50% (2)
EigenBench (three phases)	+1% (12)	+17%	+58% (2)
genome	-57% (4)	+95%	+99% (20)
vacation	-45% (8)	+79%	+92% (24)
labyrinth	-52% (24)	+5%	+67% (2)
yada	-3% (8)	+66%	+91% (22)
ssca2	-14% (24)	+11%	+62% (2)
intruder	-6% (6)	+62%	+71% (24)

Table 4.2: Performance comparison of simple model against static parallelism on applications on UMA. The higher value, the better performance.

benchmarks	best case:	average	worse case
EigenBench (one phase)	-5% (12)	+11%	+51% (2)
EigenBench (three phases)	-1% (12)	+18%	+57% (2)
genome	+6% (4)	+97%	+99% (20)
vacation	-18% (8)	+83%	+93% (24)
labyrinth	+8% (24)	+42%	+80% (2)
yada	-17% (8)	+61%	+90% (22)
ssca2	-16% (24)	+10%	+61% (2)
intruder	-31% (6)	+53%	+64% (24)

Table 4.3: Performance comparison of probabilistic model against static parallelism on applications on UMA. The higher value, the better performance

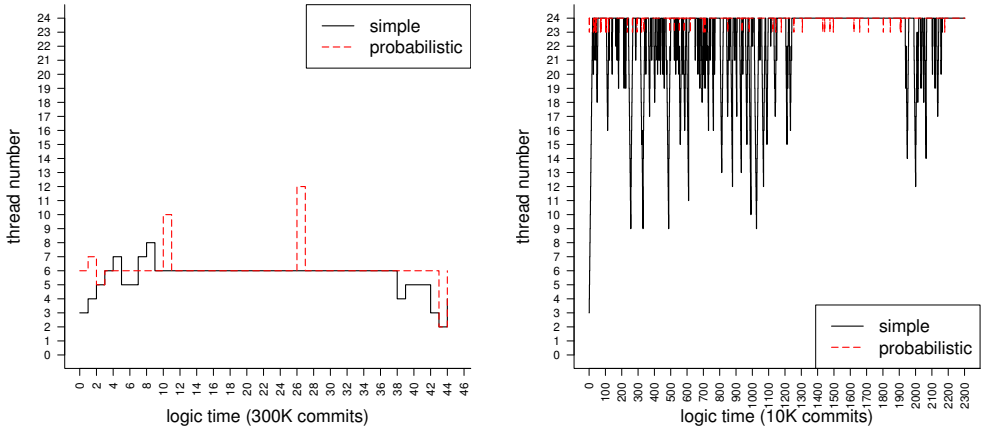
Fig. 4.15 and Fig. 4.16 elucidate the runtime parallelism variation by simple and probabilistic model. The results given are based on one execution whose performance is closest to the average execution time. As indicated in the figures, the probabilistic model gives abrupt parallelism changes contrasting with the simple model, since the probabilistic model only requires one profile length for parallelism prediction making it respond faster to CR changes. The two models show a significant difference on parallelism prediction for **EigenBench**. This is not surprising, as certain parallelism degrees generate similar throughputs in **EigenBench**. This point will be demonstrated



(a) one phase (b) three phases

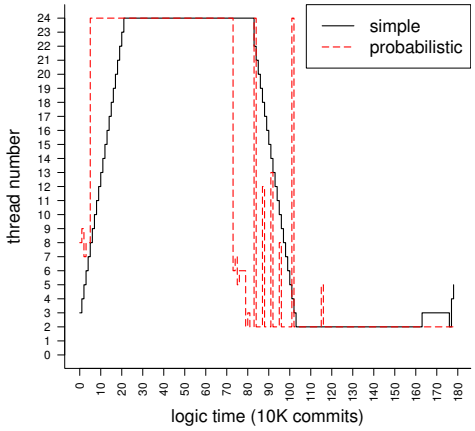
Figure 4.15: Runtime parallelism variation by the two models for **EigenBench** on UMA.

later in this section by comparing the online throughputs of diverse parallelism. A sudden parallelism degree change generated by the probabilistic model are evident on **intruder** and **genome**. For **ssca2**, the simple model gives continuous thread number changes, in contrast the probabilistic model provides relatively constant parallelism. **ssca2** has trivial runtime CR changes among all the parallelism degrees with its value close to 100%, but the computed CR range is too sensitive to performance fluctuations causing the simple model to continuously change parallelism. The probabilistic model, although reacts to predict parallelism degree frequently, often yields the maximum value. Its controller, in this case, only acts frequently in order to adjust the thread number to generate a new CR range. Both models give periodically and frequently changes of thread number on **vacation**, which alleviate performance. Such vicissitudes of parallelism degrees indicate that the CR range decided by the *simple phase detection algorithm* performs deficiently on this benchmark. **yada** has similar behaviour from 6 to 12 parallelism degrees as indicated in Fig. 4.10. Additionally, the simple model continues to regulate thread number when the throughput of an application falls within 10%. Therefore the simple model gives consistently parallelism changes from 2 to 12, while the parallelism given by the probabilistic model is more stable. **labyrinth** indicates an upwards staircase in Fig. 4.16(f), as the simple model spends relatively long time to search the optimum parallelism due to its long transaction length.

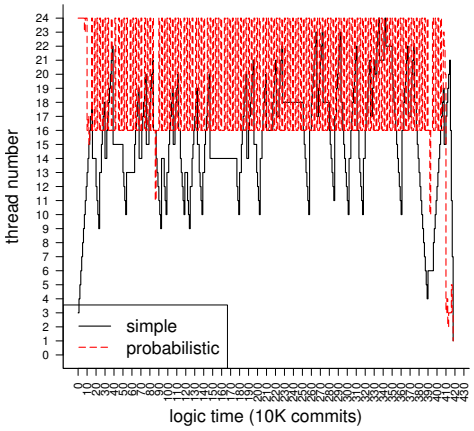


(a) **intruder**

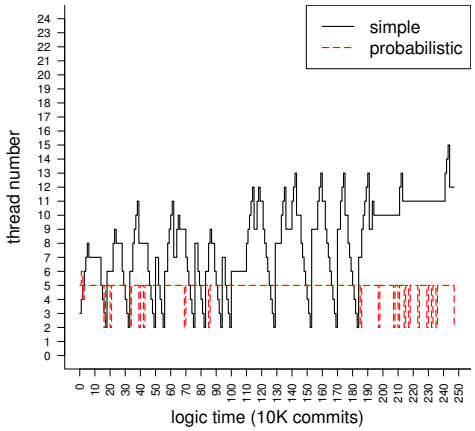
(b) **ssca2**



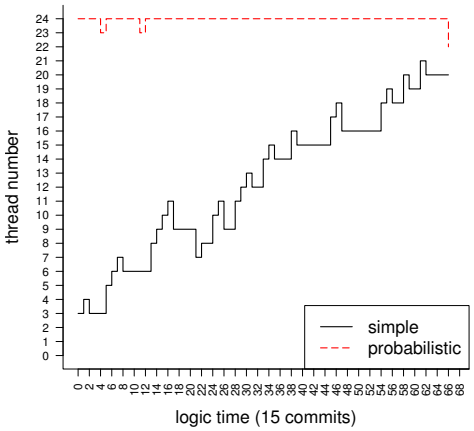
(c) **genome**



(d) **vacation**



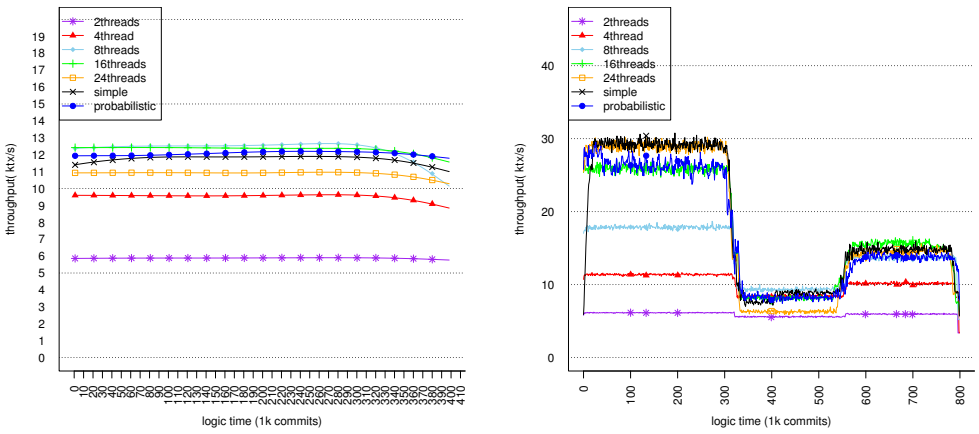
(e) **yada**



(f) **labyrinth**

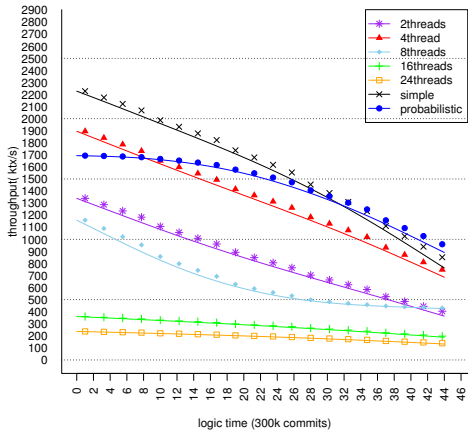
Figure 4.16: Runtime parallelism variation by the two models for **STAMP** on UMA

This paragraph discusses the correctness of the autonomic approaches. The dynamic parallelism adaptation aims at regulating the parallelism that can retain throughput at the optimum level at each phase. Ideally the throughput generated by the autonomic models should rival that generated by the static parallelism achieving the maximum throughput during entire execution. Fig. 4.17 and Fig. 4.18 elucidate the online throughput changes. To better distinguish the throughput lines generated by different parallelism, the lines in most of the figures are given in the shape of their regression curves over time, meaning that the throughputs have been smoothed. Fig. 4.17(b) and Fig. 4.18(c) illustrate the original throughput lines, as **EigenBench** (three phases) and **genome** incorporate sudden phase change. Such a change can be better illustrated by the original data. The two figures demonstrate that the online throughputs produced by the adaptive parallelism models approach the optimal throughput at each phase. In Fig. 4.17, the throughputs of certain parallelism degrees are close which provides the grounds for the thread number difference yielded by the two adaptive models for **EigenBench** in Fig. 4.15.

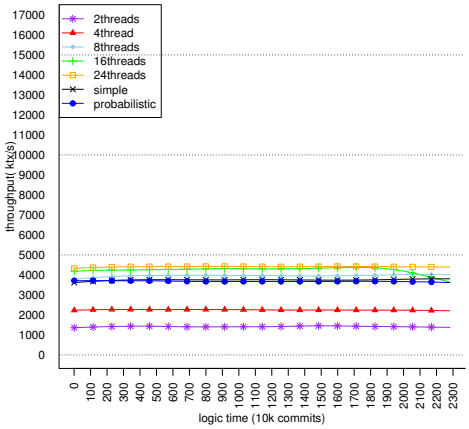


(a) one phase (b) three phases

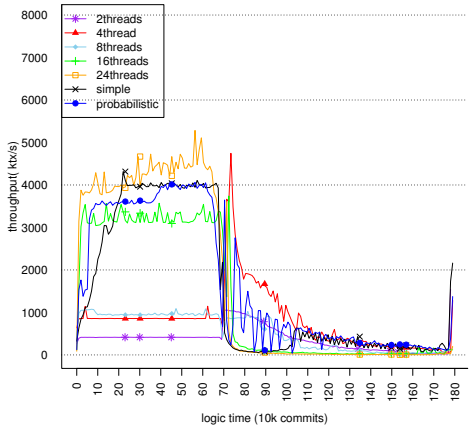
Figure 4.17: Throughput comparison for **EigenBench** on UMA. Fig. 4.17(a) presents the regression curve of the throughput over time, meaning that the throughput has been smoothed. Fig. 4.17(b) illustrates the original data to better present the clear phase change.



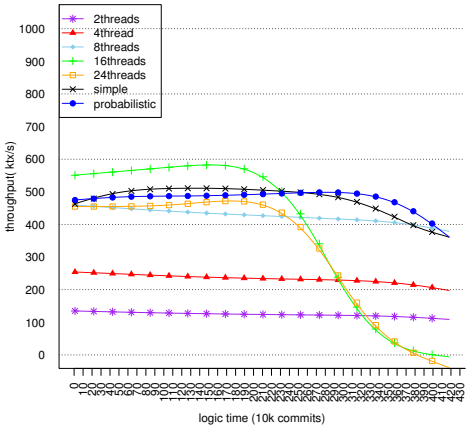
(a) **intruder**



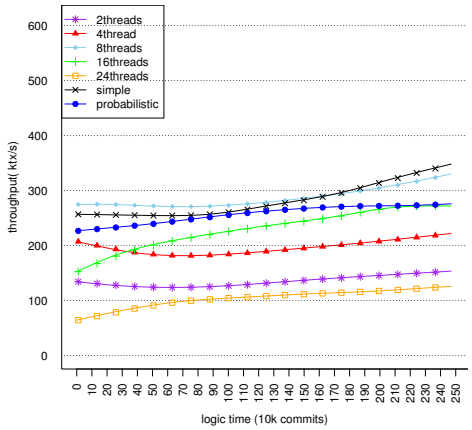
(b) **ssca2**



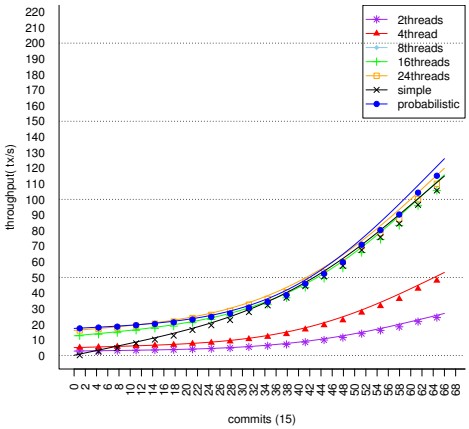
(c) **genome**



(d) **vacation**



(e) **yada**



(f) **labyrinth**

Figure 4.18: Throughput comparison for **STAMP** on UMA. Regression curves are utilised, meaning that the throughput has been smoothed.

4.5.3 Performance Evaluation on NUMA

This section presents performance evaluation on the NUMA platform. The NUMA platform incorporates distributed memory with 8 more cores than that of the UMA platform, but lower clock frequency. Therefore a change of a parallelism degree can bring higher performance impacts to an application when a remote memory access is needed. The specification of the two platforms are presented in Section 2.1, Chapter 2.

Table 4.4 and Table 4.5 indicate that the performance of the adaptive models outperform the majority of the static parallelism for all the applications except **ssca2**. Overall, the adaptive models on the NUMA platform does not perform as sufficient as on the UMA platform.

benchmarks	best case	average	worse case
EigenBench (one phase)	-11% (12)	+9%	+53% (2)
EigenBench (three phases)	0% (12)	+19%	+62% (2)
genome	-175% (4)	+79%	+97% (16)
vacation	+2% (8)	+93%	+97% (30)
labyrinth	-146% (32)	+22%	+67% (2)
yada	+13% (12)	+92%	+98% (28)
ssca2	-19% (6)	-13%	-8% (32)
intruder	+3% (6)	+70%	+82% (32)

Table 4.4: Performance comparison of simple model against static parallelism on applications on NUMA. The higher value, the better performance.

benchmarks	best case:	average	worse case
EigenBench (one phase)	-6% (12)	+13%	+55% (2)
EigenBench (three phases)	-4% (12)	+2%	+60% (2)
genome	+28% (4)	+90%	+98% (16)
vacation	-8% (8)	+92%	+96% (30)
labyrinth	+3% (32)	+52%	+87% (2)
yada	-5% (12)	+91%	+97% (28)
ssca2	-106% (6)	-94%	-86% (32)
intruder	-103% (6)	+37%	+62% (32)

Table 4.5: Performance comparison of probabilistic model against static parallelism on applications on NUMA. The higher value, the better performance

In contrast to the performance on the UMA machine, the adaptive models bring higher overhead to the application with very short transaction length (*i.e.* **ssca2**). As

illustrated in Fig. 4.12(b), **ssca2** shows similar behaviour across all the parallelism degrees with their high CR close to 100%. Therefore the probabilistic model always predicts the maximum parallelism degree as its optimum value, although it causes higher performance degradation comparing with the static maximum parallelism. Recall what is explained in Chapter 3, a monitor is utilised to control the runtime parallelism. The cost of the call to the monitor rises when the thread number increases. Such cost is significant for **ssca2** due to its very short transaction length. It suffers more from such cost on the NUMA machine. Overall, **ssca2** suffers from the performance loss by the frequent calls of the monitor and the instrumentation code. This cost is further deteriorated by a remote memory access. Nevertheless, its performance degradation is less significant on the simple model, since a lower parallelism degree is favoured by the simple model.

Analogising with the performance on the UMA machine, the probabilistic model outperforms the simple model when an application includes long transaction length and requires a high parallelism degree to achieve its optimum performance. This is the case for **labyrinth**. The simple model performs insufficiently on this application. Comparing with the performance on the UMA platform, the simple model does not even reach the maximum parallelism degree before the application execution terminates due to its slow increment of thread number.

Comparing with the performance on the UMA machine, the probabilistic model also demonstrates better performance than that of the simple model on the applications with sudden and significant parallelism changes, as in **genome**. As shown in Fig. 4.19, **genome** presents such parallelism degree changes between the first and the second phase. The program is highly contended in the second phase, making the application progress significantly slow when its parallelism degree is high. Since the simple model only decreases the thread number by one at each decision point, its performance loss is high during the transition time from the maximum parallelism to the minimum value.

Appendix A presents the runtime parallelism variation by the adaptive parallelism models for all the applications on NUMA. The online throughput comparison between the static parallelism and dynamic parallelism on NUMA is attached in Appendix B.

4.6 Discussion

Three factors give the most significant impacts on TM application performance:

1. The frequency of optimisation actions. It is also called the frequency of control

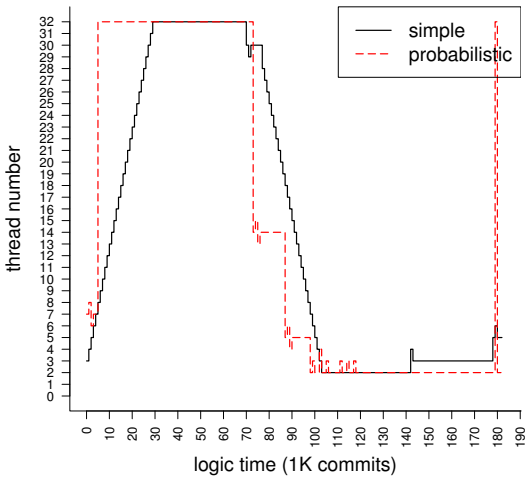


Figure 4.19: Runtime parallelism variation from the simple and probabilistic models for **genome** on NUMA.

actions. Performance penalty can occur when parallelism varies, yet is trivial on shared memory. In addition, to reduce the penalty likewise, a thread is only suspended when its current transaction commits. Nevertheless, a frequent change of parallelism can deteriorate application performance, while a slow change may not respond efficiently to program behaviour changes. The frequency of the optimisation actions mainly depends on two factors, namely the size of the profile length and the choice of the CR range. *E.g.* the CR ranges resolved for **vacation** on the UMA platform does not represent its phase difference, consequently its controller over-reacts to the CR fluctuation within the same phase.

2. The hierarchy of underlying hardware. As illustrated in Section 4.5.1, the same parallelism degree demonstrates different performance on the two platforms. For **vacation**, despite the lower clock frequency of the NUMA platform, the static optimum parallelism (8 for both machines) outperforms that on the UMA machine (around 30%). The impact of the memory architecture to applications will be better addressed in the following two chapters.

3. The precision of parallelism prediction of the probabilistic model. The probabilistic model, as its name indicated, is based on the probability theory. The model relies on two assumptions which are on the premise of the ideal situations, thus the predicted parallelism may be sub-optimum *de facto*. On the NUMA

machine for **intruder**, the probabilistic model fails to predict the near-optimum thread number in the beginning, yet produces the near-optimum parallelism degree in the later period of its execution. Parallelism prediction is more precise on the UMA platform. This is partially due to the impact of initial prediction, since it is based on the performance of 24 threads. The impact of thread migration on CR is trivial on UMA in this case, however, can be significant on NUMA.

The preceding factors can improve or deteriorate application performance. There are some other factors that attribute to the overhead imposing performance penalty. The overhead of the autonomic approaches mainly originates from four aspects:

1. Thread migration. It can introduce a large overhead especially when parallelism is adapted at runtime. Thread migration is also caused by the operation that periodically wakens a thread and suspends an active thread. Such an overhead is higher on the NUMA platform, due to its non-uniform memory access.

2. The thread number to manipulate at each parallelism profile length in the simple model. The thread number is adjusted by one only at each decision point. This causes a long parallelism profiling time making the program execute under incorrect parallelism degree.

3. The choice of throughput variation rate. The simple model keeps increasing parallelism even if the current throughput is slightly lower than the recorded maximum value in order to avoid the parallelism profiling to terminate at a regional maximum point. 10% is chosen as an acceptable variation between the current throughput and the maximum throughput.

4. The parallelism degree to start with. The probabilistic model starts with the thread number equivalent to the core number of the UMA machine, while the simple model begins with the minimum value. The former model requires a large amount of transactions executed by many threads at a fixed period to guarantee the constant probability of a conflict. A large number of threads can cause high contention. The simple model avoids the excessive contention in the beginning but hinders progress of some applications, such as **labyrinth** that need the maximum thread number to achieve its peak performance.

The simple model only adjusts the thread number by one at each decision point, thus it spends long time in searching the optimum parallelism. However this eludes

the possibility of skipping the optimum parallelism. The application starts with two threads activated rather than the maximum to avoid excessive contention which may prevent the program from progressing. This setting brings long profiling time to applications that require high parallelism. Such an overhead is difficult to compensate by the performance improvement brought by the optimum parallelism, especially when an application has a unique phase. **genome** requires the maximum parallelism during the second phase but minimal parallelism during the third phase leading to a high overhead originating from slow parallelism descending. Such an overhead is especially significant as the application is highly contented in the third phase.

The probabilistic model predicts the parallelism in one step. This diminishes the profiling time making this model present better performance than that of the simple model. However, the probabilistic model has the potential risk of overreacting to the phase variation, as this is the case for **intruder**. Lastly, the probabilistic model relies on two assumptions possibly making its prediction sub-optimum *de facto*. Thereby the simple model can outperform the probabilistic model in certain cases (*e.g.* **yada**).

4.7 Conclusion Remarks

This chapter has investigated two autonomic parallelism adaptation approaches on TinySTM (the STM platform). The system autonomicity is achieved by feedback control loops which keep auditing programs and only take actions to improve performance when necessary. This chapter firstly argues that runtime parallelism adaptation is necessary for TM applications. It then presents two approaches that can dynamically search near-optimum parallelism. The performance difference of diverse static parallelism is later examined, their results further enforce the initial argument that runtime parallelism regulation is requisite. Following that the performance of the two proposed approaches are compared with that of the static parallelism. Lastly, the implementation overhead has been analysed and advantages and disadvantages of the work have been discussed.

Apart from inappropriate parallelism, thread migration among cores impacts on the system performance and causes the performance degradation. This issue is more evident on the NUMA machine, as it has more available cores and its memory access is non-uniform. Thread migration can also impact on the precision of parallelism prediction by the probabilistic model. Performance penalty caused by thread migration can be offset by mapping threads to specific cores. Furthermore, pinning threads to

CPU cores can leverage resource utilisation, as access latency rises from the low level to a high level memory. The next two chapters will investigate the issues on thread migration and resource utilisation. New feedback control loops are designed which cooperate with the current loops to further enhance system performance.

Chapter 5

Autonomic Thread Mapping Adaptation

Chapter 5

Autonomic Thread Mapping Adaptation

5.1 Introduction

The preceding chapter raises the issues on thread migration and memory resource utilisation. The strategy of thread mapping (recall Section 2.1.1) can significantly impact on TM application performance due to data share on different memory levels. Thread mapping requires knowledge of the underlying memory architecture and interaction among threads. In addition, when design choices of the TM platform change, *e.g.* the contention manager policy changes from abort-itself to abort-others, interaction among threads often varies. Consequently, the optimum mapping strategy can vary. However, it is beyond the scope of the thesis on analysis of how thread mapping can be affected by TM designs. This dissertation is only concerned with decisions of optimum thread mapping strategy that are influenced by memory architecture, parallelism and TM applications.

Castro [5] has illustrated that the parallelism degree of an application can impact on the choice of thread mapping strategy. However, his work is limited to constant (static) parallelism degrees which poses the question on whether the mapping strategy differs when the parallelism degree varies at runtime. Such an issue will be further investigated in Chapter 6. This chapter concentrates on the influence of mapping strategies when a parallelism degree is chosen and remains constant during the whole execution. Despite of the constant parallelism degree, it is still intricate to determine a good thread mapping strategy offline for a TM application. The natural solution is to dynamically determine thread mapping strategies. Fig. 5.1 illustrates performance difference when

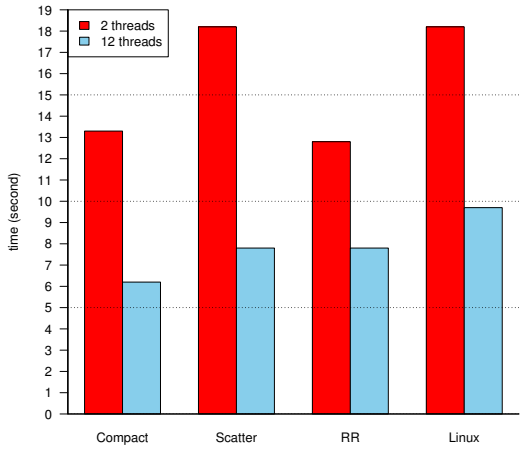


Figure 5.1: Time comparison for **yada** for four static mapping strategies.

various mapping strategies are applied on **yada** (an application of **STAMP**) executing with two and 12 threads. When **yada** runs with two threads, its optimum mapping strategy is **Compact**. The optimum strategy becomes **Round-Robin** with 12 threads. This figure manifests that the optimum thread mapping strategy can vary when an application runs with a different parallelism degree.

This chapter only analyses the dynamic thread mapping strategy when the minimum parallelism degree is selected, for the following reasons:

1. When the parallelism degree reaches the number of available cores, the mapping strategy no longer gives impact to application performance.

2. Some TM applications present unstable behaviour when running with a high parallelism degree. Hence there is little interesting to investigate performance impact from thread mapping when the parallelism degree is high.

3. This chapter focuses on the performance impacts originating from thread mapping, it does not intend to list performance diversity when thread mapping is applied to all the possible parallelism degrees.

In this thesis, the *static thread mapping strategy* means that the thread mapping strategy is chosen before program execution and remains constant at runtime, while the *dynamic thread mapping strategy* means the strategy is selected during execution and may vary afterwards.

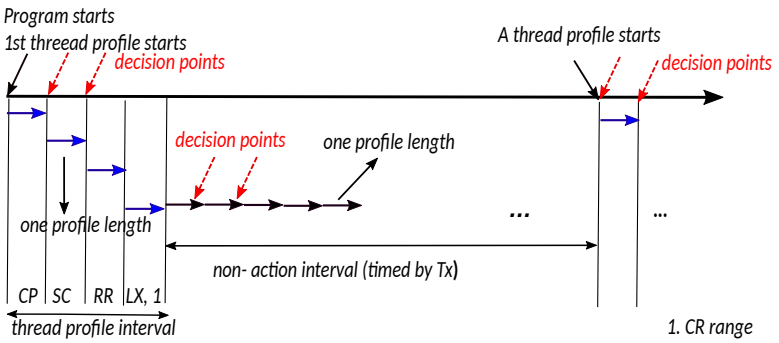
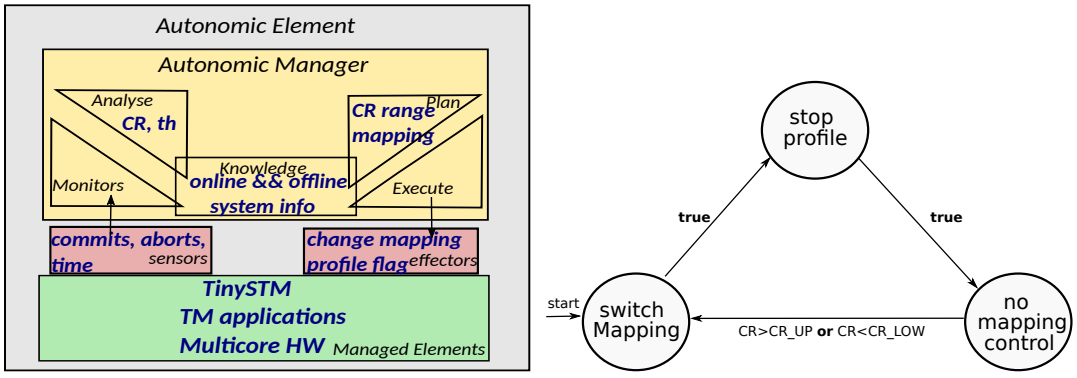


Figure 5.2: Profiling procedure for obtaining the optimum thread mapping strategy.

This chapter firstly describes the dynamic thread mapping strategy in Section 5.2. Next, Section 5.3 presents performance evaluation on static and dynamic thread mapping strategies for the UMA and NUMA machines. Following that, Section 5.4 discusses the pros and cons of the proposed approach. Section 5.5 concludes this chapter.

5.2 Dynamic Thread Mapping

A feedback control loop is constructed to consecutively improve the decision on the thread mapping strategy. As illustrated in Fig. 5.2, each strategy is profiled during the thread profile interval. A new strategy is switched at each decision point. The program enters into an non-action interval when all the strategies have been profiled. The feedback control loop is fairly simple for dynamic thread mapping control as depicted in Fig. 5.3.



(a) The MAPEK-shape feedback control loop. (b) The structure of autonomic manager

Figure 5.3: The feedback control loop for dynamic thread mapping control. *th* and *mapping* stand for thread number and thread mapping strategies, respectively.

5.2.1 Inputs and Outputs

Under control terminology, the inputs of the feedback control loop is commit, abort and physical time (see Section 2.3.1 and Section 3.2.1 for definition). The outputs/control actions are switching thread mapping strategies and setting the profile flag.

5.2.2 Decision Functions

The feedback control loop for thread mapping control incorporates three decision functions: the *CR range decision function*, the *profile decision function* and the *thread mapping strategy decision function*. The advance phase-detection algorithm is utilised in this chapter as the CR range decision function. The other two decision functions and their interaction with the CR range decision function are described as an automaton. The automaton (Fig. 5.3(b)) is possessed of three states.

To determine the optimum thread mapping strategy from a set of strategies for an application, each strategy is profiled for one profile length and the one that yields the highest throughput is selected as the optimum strategy. This procedure corresponds to the state ***switch mapping***. Every thread interval starts with the **Compact** strategy, and continues with **Scatter**, **Round-Robin** (if applicable) and terminates with **Linux**. The ***stop profile*** state decides the CR range and sets the optimum mapping strategy. The automaton shifts to ***no mapping control***, where the controller only audits if the program stays in the same phase. When CR falls out of the CR range, the automaton transitions into the ***switch mapping*** state again and a new thread profile interval starts.

5.3 Performance Evaluation

This section presents performance diversity for static thread mapping strategies and dynamic thread mapping strategy for **EigenBench** and **STAMP**.

5.3.1 Static Thread Mapping

Figs. 5.4, 5.5 and 5.6 illustrate performance difference on the UMA and NUMA machines. Recall that, this thesis describes 4 thread mapping strategies based on cache share and Linux default thread scheduling. The NUMA machine is not eligible for setting **Round-Robin**, as its L2 cache possesses only one core. **Round-Robin** is applicable for the memory architecture whose L2 cache is shared by no less than two

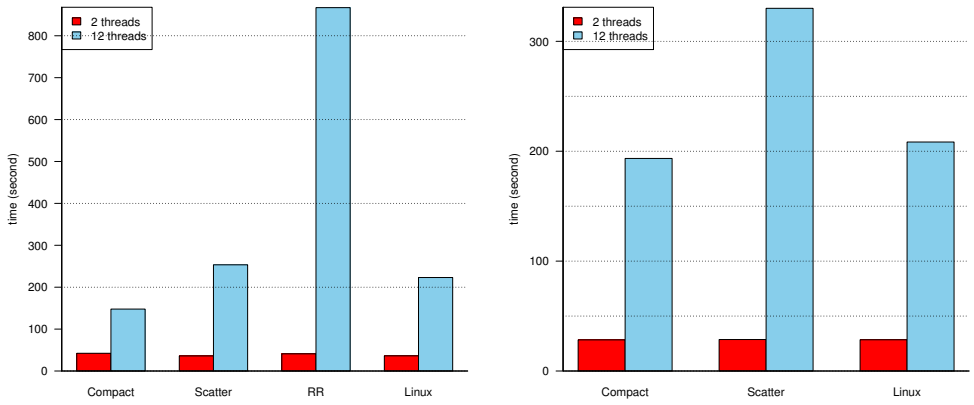
cores. The figures only present the results for the minimum parallelism degree (two) and the parallelism degree equivalent to half the core number (12 on UMA, 16 on NUMA). Some conclusions can be reached from the figures:

1. The impact of parallelism degree is often more significant than that of the mapping strategy.

2. The performance difference from the four strategies is insignificant on some applications, *e.g.* **labyrinth**, regardless of the thread number.

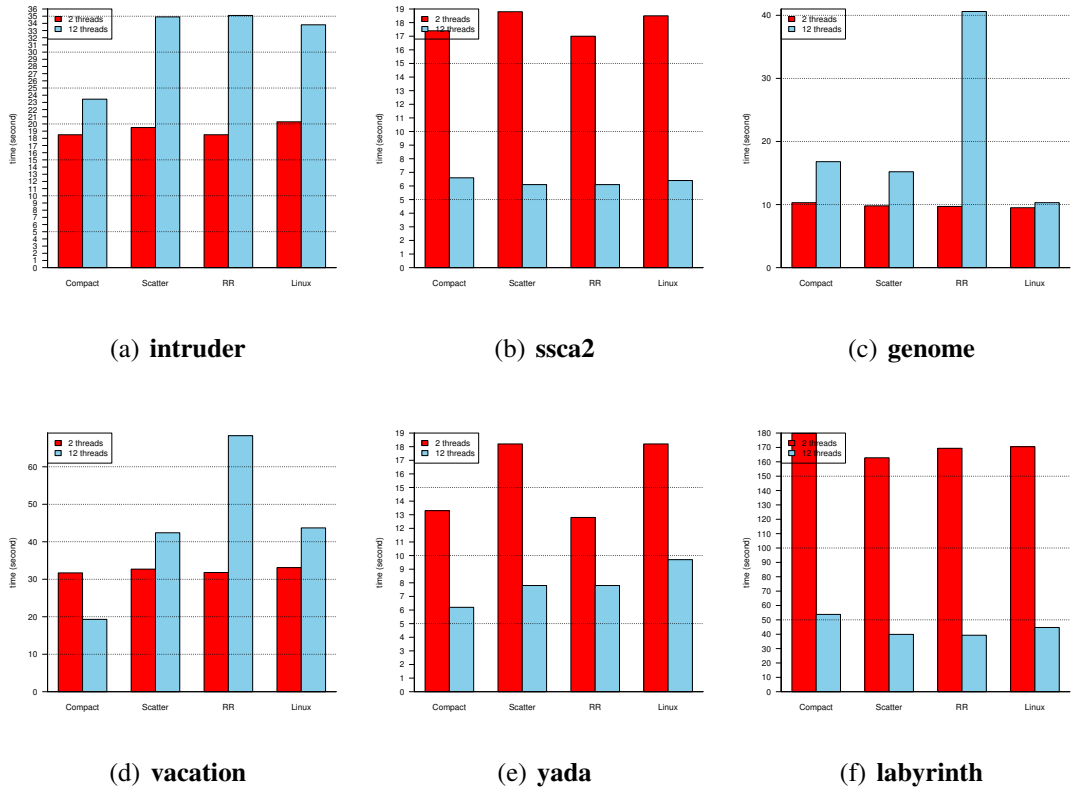
3. Some thread mapping strategies perform similarly on certain applications. *E.g.* **Scatter** demonstrates analogous behaviour as **Rounb-Robin** when half of the core number is utilised **yada** as shown in Fig. 5.5(e). When the minimum parallelism degree is applied, **Compact** and **Round-Robin** gives similar performance. The impact of mapping strategies is not obvious on **genome** when the minimum parallelism degree is chosen.

4. The affect of mapping strategies on applications varies between the UMA and NUMA machines. *E.g.* when the parallelism degree of **genome** is equivalent to the core number, on the UMA machine **Compact** outperforms the other strategies, in contrast **Scatter** performs the best on the NUMA machine.



(a) UMA (b) NUMA

Figure 5.4: Time comparison of **EigenBench** (two phases) for static mapping strategies on UMA and NUMA. **Round-Robin** is not applicable on the NUMA platform.



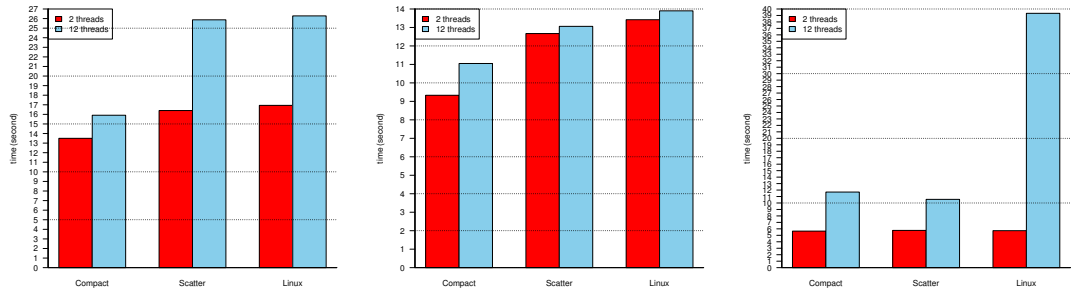
(a) **intruder** (b) **ssca2** (c) **genome**

(d) **vacation** (e) **yada** (f) **labyrinth**

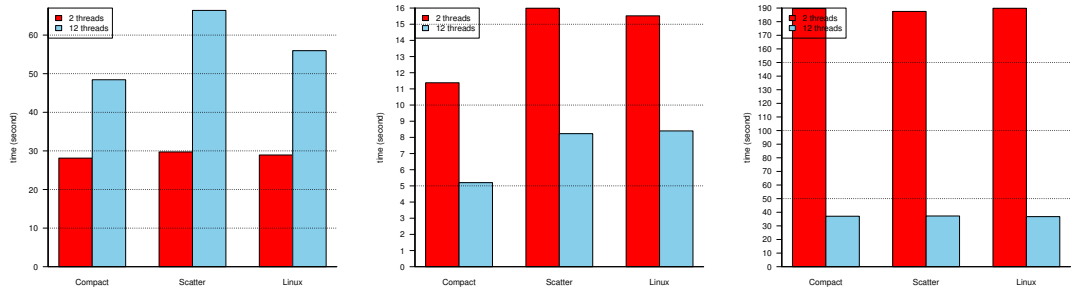
Figure 5.5: Time comparison for **STAMP** for static mapping strategies on UMA.

5.3.2 Performance Comparison on Static and Dynamic Mapping

The previous section indicates that dynamic thread mapping is necessary, as it is difficult to dictate a mapping strategy offline that can still benefits an application when its parallelism degree alters. This section presents result evaluation of the dynamic strategy and compares it with that of the static strategies. It demonstrates that online adaptation of mapping strategies is needed. In the measurements visualised in Fig. 5.7, dynamic mapping outperforms any static strategy on **EigenBench** (two phases). This application of **EigenBench** incorporates two phases. The first phase chooses **Scatter** as its best mapping strategy, whereas the second phase favours **Compact**. Fig. 5.8 and Fig. 5.9 elucidate the performance comparison for **STAMP**. The majority of **STAMP** applications manifest performance degradation against the best static mapping strategy. Most of **STAMP** applications do not exhibit the necessity of dynamic mapping profiling. It is sufficient enough to profile once and apply the decision during the remaining execution, therefore penalty caused by profiling mapping strategies can not be offset



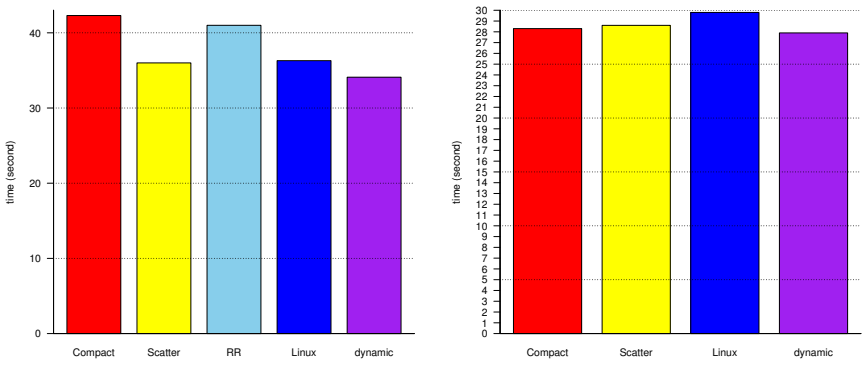
(a) **intruder** (b) **ssca2** (c) **genome**



(d) **vacation** (e) **yada** (f) **labyrinth**

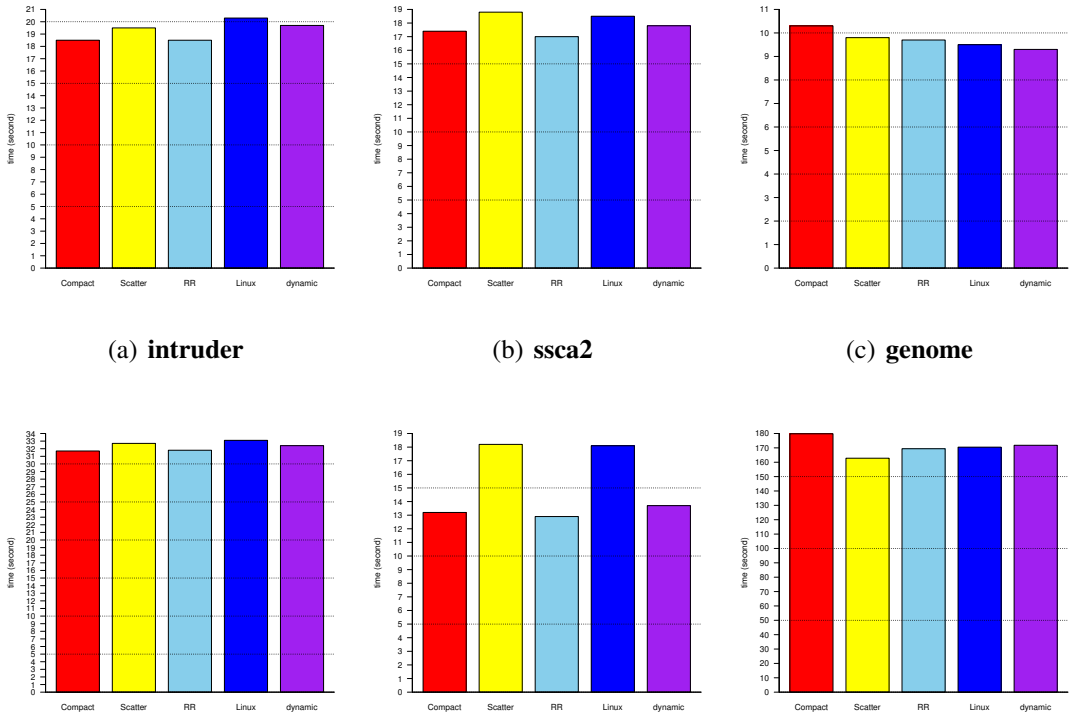
Figure 5.6: Time comparison for **STAMP** for static mapping strategies on NUMA. **Round-Robin** is not applicable on the NUMA platform.

by its gain. A notable exception that the dynamic approach benefits **STAMP** applications is **genome**. As analysed in Section 4.5.2, **genome** incorporates two main phases at runtime. The first phase only possesses read operations, while the second phase includes both read and write operations. Hence the optimum strategy is predicted twice: **Round-Robin** (on UMA) or **Scatter** (on NUMA) is chosen for the first phase, **Compact** is selected for the second phase. However its speed-up is negligible on the UMA machine. As illustrated in Fig. 5.5(c), four strategies offer performance difference in a small degree. *Per contra*, on the NUMA machine, the performance difference is negligible when the three strategies are applied (this is partially due to shorter execution time on the NUMA). Thus the dynamic approach only benefits **genome** on the UMA platform.



(a) UMA (b) NUMA

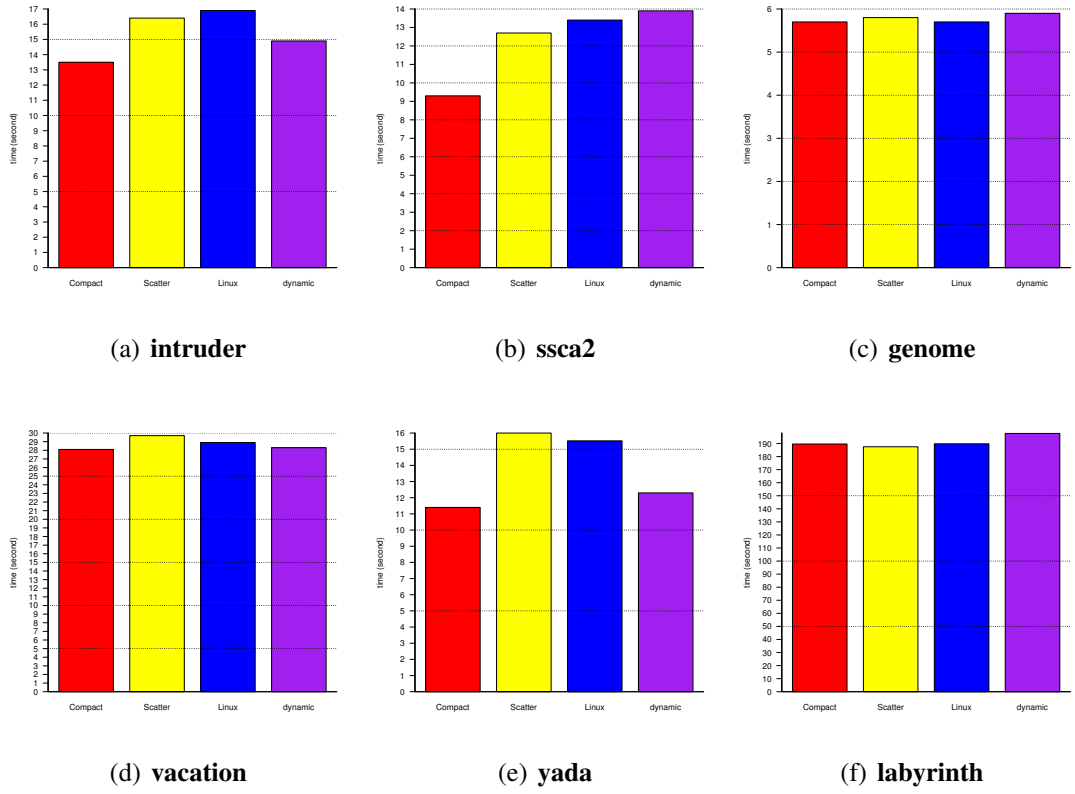
Figure 5.7: Time comparison of **EigenBench** (two phases) for static and dynamic mapping strategies on UMA and NUMA. **RoundRobin** is not applicable on the NUMA platform.



(a) **intruder** (b) **ssca2** (c) **genome**

(d) **vacation** (e) **yada** (f) **labyrinth**

Figure 5.8: Time comparison for **STAMP** for static mapping strategies on UMA.



(a) **intruder** (b) **ssca2** (c) **genome**

(d) **vacation** (e) **yada** (f) **labyrinth**

Figure 5.9: Time comparison for **STAMP** for static mapping strategies on NUMA. **RoundRobin** is not applicable on the NUMA platform.

5.4 Discussion

The overhead of the dynamic thread mapping approach mainly originates from:

1. Time wasted in the wrong mapping strategy during profiling. In order to detect the optimum strategy, each strategy is applied for one profile length making the program execute under non-optimum setting. Therefore, only the application with diverse phases can benefit from this approach.

2. Thread migration. It results in thread migration to switch between two thread mapping strategies. The more often the strategy changes, the higher performance penalty can be caused.

In Section 5.3.1, the performance evaluation on diverse static mapping strategies indicates that the best strategy can vary when a new parallelism degree is applied. Section 5.3 demonstrates that changing mapping strategies dynamically can improve program performance when two threads are executing. However, the dynamic approach

only shows performance gain on two applications against the best static strategy.

5.5 Conclusion Remarks

This chapter firstly introduces the necessity of dynamic thread mapping, then it presents an approach that can dynamically adjust mapping strategies. Following the description of this approach, performance evaluation is given on comparison of static and dynamic strategies. Lastly, the implementation overhead is discussed.

The conclusion has been made that it is intricate to detect the best thread mapping strategy for an application offline. This chapter illustrates that the best mapping strategy can vary when a different parallelism degree is chosen for the same application. The performance difference can also differ when the same application executes on a difference platform. Performance diversity has been examined from four thread mapping strategies (only three on NUMA) and the optimum strategy can alter in different applications.

Through the presentation of this chapter and the preceding chapter (Chapter 4), one conclusion can be reached: it can bring performance improvement by either dynamically adapting parallelism or thread mapping strategies. This raises one question on whether performance can be further improved by coordinating the two dynamic approaches? The next chapter will offer insights into this issue.

Chapter 6

Coordination of Thread Parallelism and Thread Mapping Adaptation

Chapter 6

Coordination of Thread Parallelism and Thread Mapping Adaptation

Chapter 4 and Chapter 5 have demonstrated that parallelism degrees and thread mapping strategies can both impact on application performance. It has been manifested that dynamic parallelism or thread mapping adaptation is feasible and necessary. As illustrated in Fig. 5.1, the optimum mapping strategy differs when the parallelism degree alters. Coordination of thread parallelism and mapping adaptation, thereby, becomes necessary.

The remainder of this chapter firstly analyses the complexity and obstacles of managing both parallelism degree and thread mapping at runtime in Section 6.1. To simplify the description, the control of both thread parallelism and thread mapping is addressed as *thread control* in the rest of this thesis. The overview of the algorithm is presented in Section 6.2 and is detailed in Section 6.3. Following that, Section 6.4 depicts performance evaluation. Advantages and disadvantages are entailed in Section 6.5. Section 6.6 concludes this chapter.

6.1 The Complexity of Dynamic Parallelism and Mapping Control

Application performance affected by thread mapping can be measured by cache miss ratio and cache hit ratio, as a good mapping strategy improves cache usage. However, this thesis does not utilise cache information to indicate performance for the examined thread mapping, as the reasons stated below:

- Cache misses can be caused by round-robin thread scheduling. Round-robin scheduling (See Section 3.3.2) guarantees fair execution time among threads, but it causes cache misses for its working principle: the suspended threads are invoked and the active threads are suspended periodically, consequently causing thread migration and inducing cache misses. These cache misses are unpreventable and impact on measurement of the cache misses originating from data accessing by threads.

- Threads need to remap to cores, when the parallelism predictor gives new prediction. It is difficult to distinguish this type of cache misses from the others.

- The L3-level cache information is not always accessible by users, *e.g.* the UMA platform utilised for performance evaluation in this thesis does not provide the interface for L3-level cache access.

Hence the same metrics as in the proceeding two chapters are utilised, *i.e.* CR and throughput, to indicate runtime performance of thread mapping.

6.1.1 The Threshold of Parallelism Degree for Thread Mapping

As elucidated by Fig. 5.5 in Chapter 5, the best thread mapping strategy can differ when the parallelism degree varies. When the active thread number is equivalent to the core number, there is little interest to choose thread mapping strategies. Additionally, some TM applications experience significant performance degradation when their parallelism degree increases, or their performance becomes unstable. Therefore thread mapping is less interesting for such applications running with a high parallelism degree. Based on the above reasons, the author chooses half of the core number as the threshold for activating control of thread mapping strategies. When the parallelism degree surpasses this value, the application utilises the default mapping strategy (**Linux**).

6.1.2 The Frequency of Thread Mapping Prediction

The cost of setting a thread mapping strategy can be high, therefore the frequency of adjusting the strategy should be low. The frequency of switching thread mapping strategies brings non-trivial impacts on application performance, as the improvement of memory resource utilisation obtained by thread mapping can be forfeited by its associated potentially high cost of thread migration.

Setting a threshold as aforementioned in Section 6.1.1 reduces mapping frequency. To further alleviate the cost of thread migration, yet ensure the controller to be responsive to phase changes, the control actions of changing mapping strategies are only invoked if the change of parallelism degree exceeds four. This is a tuning parameter, and application performance can be affected by adjusting this value. Furthermore, no mapping is triggered the parallelism degree keeps the same value as the one before its prediction. Since when the parallelism stays unmodified, the program phase remains unchanged.

6.1.3 The Order of Decision Making

Two control decisions need to be taken: thread parallelism degree and thread mapping strategy. This brings up the question on which decision should be made first. As demonstrated in the preceding chapter, the parallelism degree can affect the choice of the best thread mapping strategy. Intuitively, the thread mapping strategy may in turn affect the parallelism prediction. This thesis presents the algorithm which chooses to predict parallelism prior to mapping under the scrutiny as stated below:

1. The prediction of the thread mapping strategy requires knowledge of the parallelism degree. Some TM applications do not scale with an increase of its parallelism degree regardless of the mapping strategy that is applied, *e.g.* **genome** with the 12 thread number. It is unnecessary to predict the mapping strategy when the behaviour of a program is unstable.

2. The parallelism degree demonstrates more significant performance impacts than that of the thread mapping strategy.

3. The frequency of thread mapping decision depends on the parallelism degree.

4. The impact of the thread mapping strategy on parallelism prediction is trivial. This is based on observation of application performance.

6.2 Overview of the Profiling Procedure

Recall that Chapter 4 describes two models for near-optimum parallelism prediction: the simple model and probabilistic model. This section only utilises the latter model to predict parallelism for the following reasons:

1. The time spent for parallelism prediction is short. The simple model responds slowly to program phase changes and requires longer time for detection. *Per contra*, the probabilistic model responds rapidly to phase variation and needs short parallelism profile time despite that it may over-react to phase fluctuations.

2. It shortens the thread profile interval. The control loop described in Chapter 5 needs three or four profile lengths to decide the best mapping strategy. Combining the simple model with the thread mapping method, the thread profile interval is *de facto* at risk to exceed the length of a program phase.

Fig. 6.1 illustrates the profiling procedure for prediction of thread parallelism and mapping. In the first decision point during a thread profile interval, the parallelism is predicted and is subsequently verified in the second decision point. The profiling procedure may proceed with four (three on the NUMA) profile lengths to analyse each thread mapping strategy. The procedure of profiling mapping strategies is the same as stated in Chapter 5, however, whether to trigger this procedure depends on the conditions described in Section 6.1.1 and Section 6.1.2. The last decision point sets the optimum thread mapping strategy and parallelism degree as well as the CR range. After this thread profile interval, the program remains in the non-action interval where its performance is monitored and a new thread profiling procedure is scheduled when the program enters a new phase.

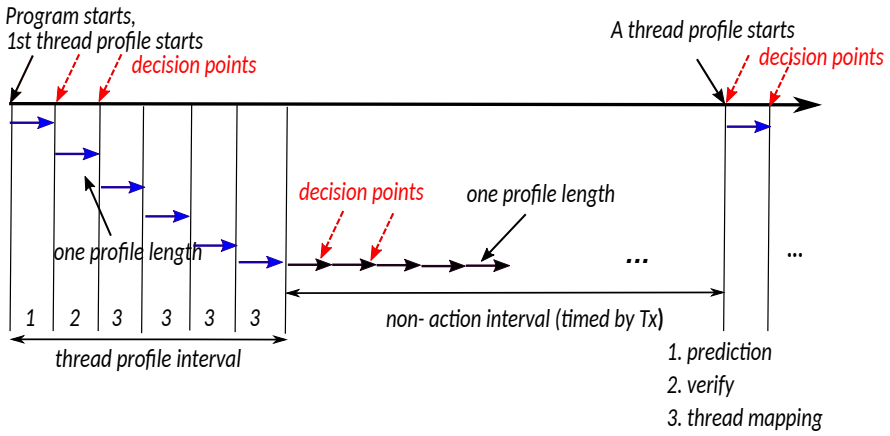


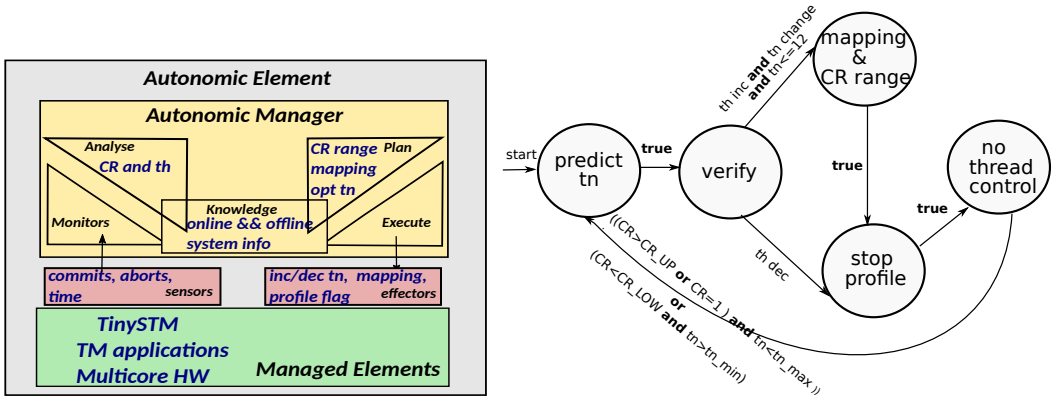
Figure 6.1: Periodical profiling procedure for thread control. At each decision point (marked by dashed red arrow), control actions are taken. One profile length is a fixed number of commits.

6.3 Control Coordination

6.3.1 Inputs and Outputs

The inputs of the loop are commits, aborts and physical time (see Section 2.3.1 and Section 3.2.1 for definition). The outputs/actions are increasing or decreasing parallelism degree, changing thread mapping strategy and setting the profile flag.

6.3.2 Coordination of Control Loops



(a) The instantiation of MAPE-K-shape feedback control loop.

(b) The structure for the autonomic manager of Fig.6.2(a) in an automaton shape.

Figure 6.2: The feedback control loop. th, tn and mapping stand for throughput, thread number and thread mapping strategy, respectively. One state corresponds to one decision point in Fig. 6.1. The boolean value **true** means the unconditional state shift.

Fig. 6.2(a) presents the MAPE-K loop for thread control. The autonomic manager is depicted in Fig. 6.2(b) as an automaton. It is composed of five states. Four decision functions cooperate to make decisions. A *parallelism predictor*, a *thread mapping strategy decision function*, a *CR range decision function* (decides the phase change) and a *thread profile decision function* (enables/disables the thread profile action). This section only describes the thread profile decision function and its relationship with other functions. The first two functions have been detailed in the previous two chapters. The advanced-phase detection algorithm presented in Chapter 3 (Page 42) is utilised as the CR range decision function.

The automaton as illustrated in Fig. 6.2(b) starts in the ***predict tn*** state where the optimum parallelism is predicted. Having executed the predicted parallelism for one profile length, the automaton unconditionally shifts to the ***verify*** state which corresponds

to the second decision point in Fig. 6.1. In the ***verify*** state, the predicted optimum thread number is verified by comparing the current throughput with the throughput before the prediction. If the current throughput is higher than that value, the predicted thread number is kept, otherwise the parallelism degree is switched back to the previous optimal value. Additionally, the ***verify*** state decides the necessity of profiling the thread mapping strategy, *i.e.* it is only profiled when the following conditions are both satisfied:

1. The thread number has been changed (as reasoned in Section 6.1.2).

2. The new thread number does not exceed half of the core number (as reasoned in Section 6.1.1).

When the above conditions are met, the automaton shifts to the ***mapping & CR range*** state where each thread mapping strategy is profiled, meanwhile a new CR range is computed. Otherwise, the automaton transitions to the ***stop profile*** state suspending all the thread regulation. The automaton transitions from the ***mapping & CR range*** state to ***stop profile*** unconditionally. At the final decision point of a thread profile interval, the parallelism and thread mapping strategy are set to the values which yield the maximum throughput. At each decision point of a non-action interval which corresponds to the ***no thread control*** state, the profile decision function decides if a new thread profile is needed. A new thread profile procedure always starts from the ***predict tn*** state. It is worth noting that the profile decision function is performed on the ***verify***, ***mapping & CR range*** and ***no thread control*** state.

The coordination of the four decision functions are illustrated in pseudocode in Fig. 6.3 to further clarify their integration and to illustrate their implementation. The profile decision function is composed of two parts: disable and enable profile actions.

6.4 Performance Evaluation

This section presents the results of autonomic thread parallelism and mapping adaptation on the UMA and NUMA machines. Its results are also compared with the ones generated by adaptation only on parallelism. The approach that only adjusts parallelism is addressed as *dynamic parallelism model*, while the method that regulates both parallelism and thread mapping strategy is addressed as *dynamic thread control model*. Since this chapter utilises the advanced phase detection algorithm, the *dynamic*

```
/*CR range decision func and parallelism predictor*/
 thread_prediction_func(){...}//see previous derivation
 CR_range_func(){...}//see previous derivation
```

```
/*thread mapping and disable profile decision funcs*/
thread_mapping_prediction()
{
  for i in {Compact,Scatter,RR,Linux} //mapping strategy
     apply i;
     if (current throughput> max throughput)
        optimum mapping= i;
  apply optimum mapping;
}
if (current throughput > max throughput)
    compute a new CR range;
    max throughput=current throughput;
    if (optimum tn <= half the core number && tn changes )
       apply thread mapping prediction;
    else
       disable profile;
else
    optimum tn = previous tn;
    apply previous optimum thread mapping;
    disable profile ;
```

```
/*enable profile decision fuc*/
 enable_profile_action()
{
  record current mapping strategy;
  record current max throughput;
  record current tn;
}
if CR falls out the CR range
    enable thread profile action;
else
    keep collecting profile info;
```

Figure 6.3: The implementation of the four decision functions. tn and thread mapping stand for the thread number and thread mapping strategy, respectively.

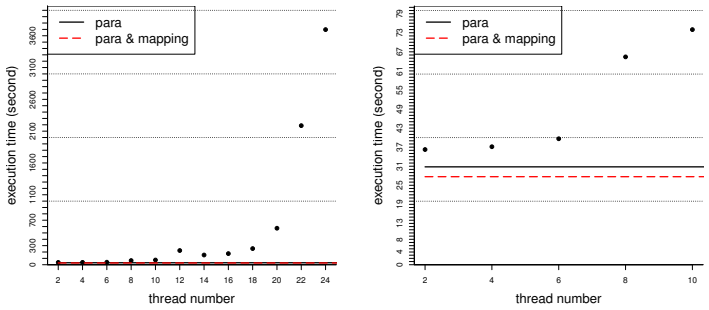
parallelism model also employs this algorithm for phase detection as comparison. Recall that Chapter 4 presents the probabilistic parallelism prediction model that utilises the simple phase decision algorithm. Therefore, the performance of the autonomic parallelism adaptation in this section differs from that in Chapter 4. It is worth noting that this thesis is only concerned with parallel applications, hence the results of sequential applications are not presented.

6.4.1 Results on the UMA Machine

This section firstly presents the execution time comparison between the two dynamic models as well as static parallelism. It then illustrates the runtime changes of thread parallelism and mapping strategy. Lastly, in order to validate the dynamic approaches

on prediction of suitable parallelism and mapping strategies at each phase, the comparison of online throughput variation is given.

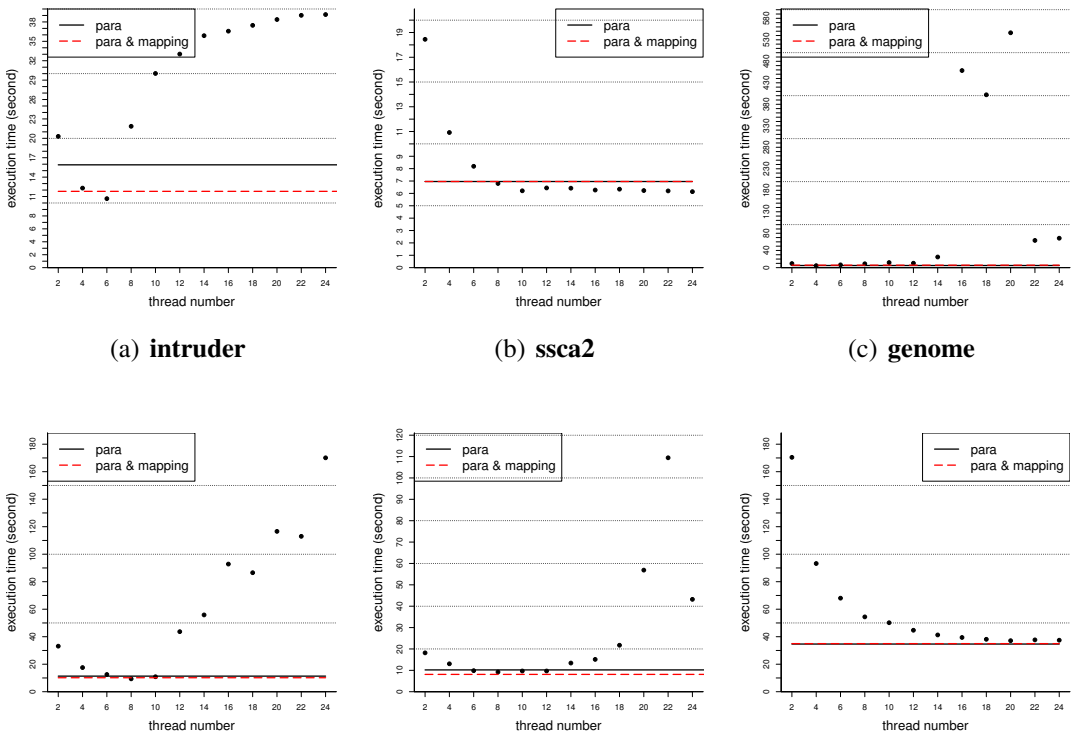
Fig. 6.4 and Fig. 6.5 illustrate the execution time comparison with different static parallelism, and two dynamic models of **EigenBench** and **STAMP** on the UMA machine. The dots represent the execution time with different static parallelism degrees from two to the core number (24). The solid black line stands for execution time obtained with the *dynamic parallelism model* and the dashed red line gives the execution time with the *dynamic thread control model*. Fig. 6.4(b) only compares the performance of up to 10 threads to better illustrate the performance difference between two dynamic models on **EigenBench**.



(a) all threads (b) up to 10 threads

Figure 6.4: Time comparison of **EigenBench** for static parallelism, *dynamic parallelism model* and *dynamic thread control model*. The dots represent the execution time with different static parallelism.

As can be seen in Fig. 6.4 and Fig. 6.5, the two models outperform the majority of the static parallelism degrees. The *dynamic thread control model* shows positive performance rise against the *dynamic parallelism model* on the applications, *i.e.* **EigenBench**, **yada** and **intruder**, but it indicates performance degradation on **genome** and **vacation**. Both models perform similarly on **labyrinth** and **ssca2**. Table 6.1 and Table 6.2 detail the performance comparison. The digits in the brackets are the static parallelism which gives the best and the worst performance, respectively. The symbol "plus" ("+") means performance gain against the compared value. Both models outperform the best case of static parallelism on **EigenBench**, **genome** and **labyrinth**. Both models show performance degradation on **vacation**, **intruder** and **ssca2** against the best case, yet significantly improve performance compared with the average value and the worst case. It is worth noting that, some TM applications scale poorly when



(a) **intruder** (b) **ssca2** (c) **genome**

(d) **vacation** (e) **yada** (f) **labyrinth**

Figure 6.5: Time comparison for **STAMP** for static parallelism, *dynamic parallelism model* and *dynamic thread control model* . The dots represent the execution time with different static parallelism.

their parallelism rises (*e.g.* **genome**, **vacation**, **EigenBench**) due to TM mechanisms on transaction conflict detection and resolution.

benchmarks	best case	average value	worse case
EigenBench (two phases)	+15% (2)	+95%	+99% (24)
genome	+16% (4)	+97%	+99% (20)
vacation	-13% (8)	+83%	+94% (24)
labyrinth	+7% (24)	+42%	+80% (2)
yada	-11% (8)	+63%	+91% (22)
ssca2	-4% (24)	+19%	+65% (2)
intruder	-48% (6)	+46%	+59% (24)

Table 6.1: Performance comparison of different applications with *dynamic parallelism model*. The performance is compared with the minimum, average and the maximum value of all the static parallelism.

Fig. 6.6 and Fig. 6.7 demonstrate the parallelism variation for both models and the

benchmarks	best case	average value	worse case
EigenBench (two phases)	+24% (2)	+96%	+99% (24)
genome	+2% (4)	+96%	+99% (20)
vacation	-50% (8)	+78%	+91% (24)
labyrinth	+6% (24)	+41%	+80% (2)
yada	+12% (8)	+70%	+93% (22)
ssca2	-4% (24)	+19%	+65% (2)
intruder	-10% (6)	+60%	+70% (24)

Table 6.2: Performance comparison of different applications with the *dynamic thread control model*. The performance is compared with the minimum, average and the maximum value of all the static parallelism

mapping strategy variation for the *dynamic thread control model*. The gap between two vertical blue lines is the phase transition region. To better illustrate the performance impact of thread mapping strategies, the results presented in the two figures are the best results out of 10 executions. As shown in the figures, the **EigenBench** application shows thread mapping strategy variation between **Scatter** and **Compact**. In contrast, the majority of applications in **STAMP** keep the same strategy during their entire execution. **genome** illustrates the thread mapping strategy change at runtime. This is due to the fact that the mapping strategy is not profiled in its first phase, as the maximum parallelism degree is predicted leading to the selection of the default mapping strategy. Likewise, **vacation** changes its thread mapping strategy at the second phase when its parallelism degree decreases significantly.

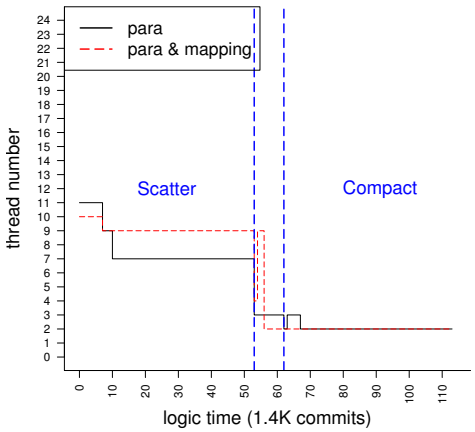
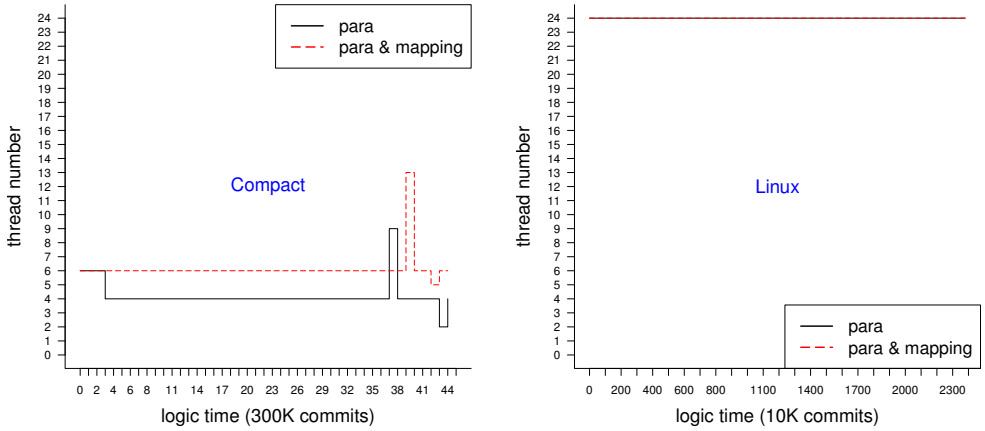
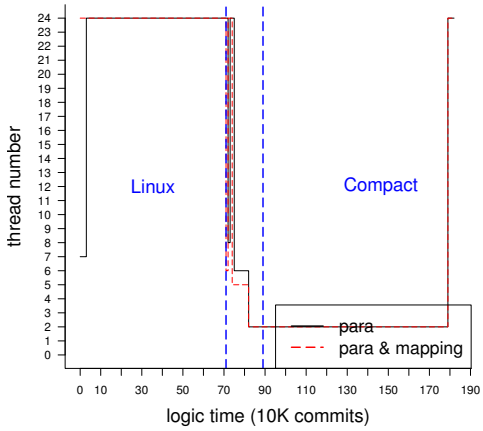


Figure 6.6: Runtime variation of parallelism and mapping strategies by the two models for **EigenBench** on UMA.

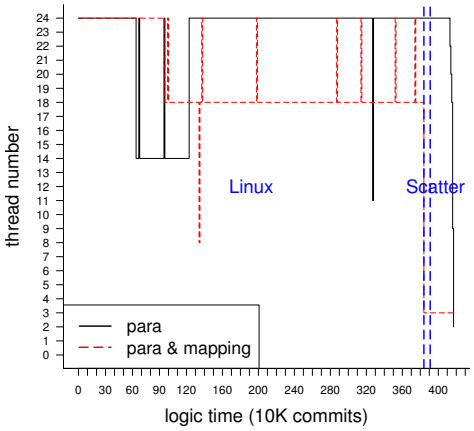


(a) **intruder**

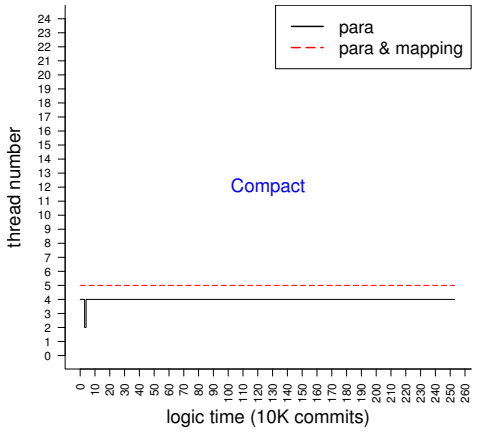
(b) **ssca2**



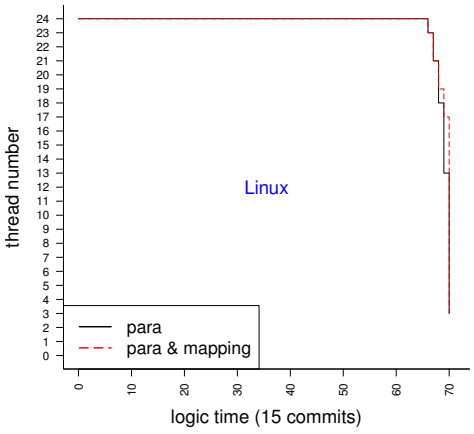
(c) **genome**



(d) **vacation**



(e) **yada**



(f) **labyrinth**

Figure 6.7: Runtime variation of parallelism and mapping strategies by the two models for **STAMP** on UMA

Lastly, Fig. 6.8 and Fig. 6.9 show the online throughput comparison on the UMA machine. To better distinguish the throughput lines generated by different parallelism, the lines in most of the figures are given in the shape of their regression curves over time, meaning that the throughputs have been smoothed. Only Fig. 6.9(c) and Fig. 6.8 illustrate the original throughput lines, as **EigenBench** (two phases) and **genome** incorporate sudden phase changes, it can better illustrate the phase transition by presenting the original data. According to the figures, both models converge or exceed the maximum throughput of the static parallelism in each phase.

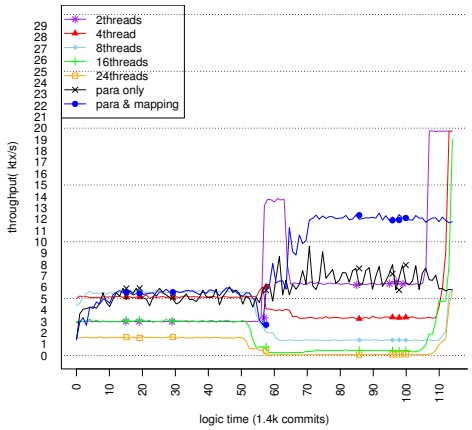
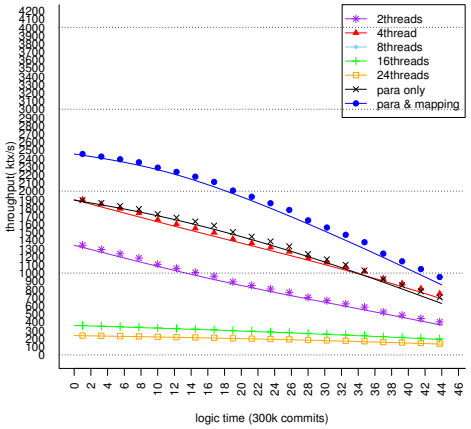
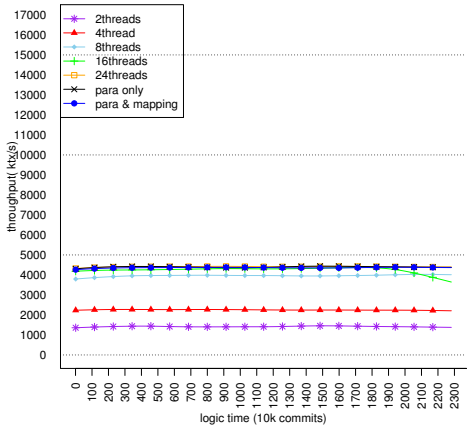


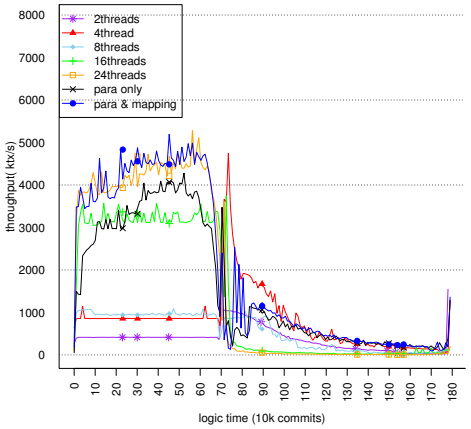
Figure 6.8: Time comparison of **EigenBench** (two phases) for static parallelism, *dynamic parallelism model* and *dynamic thread control model* on UMA. The dots represent the execution time with different static parallelism.



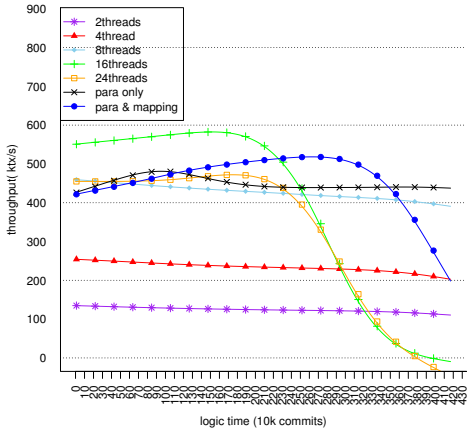
(a) **intruder**



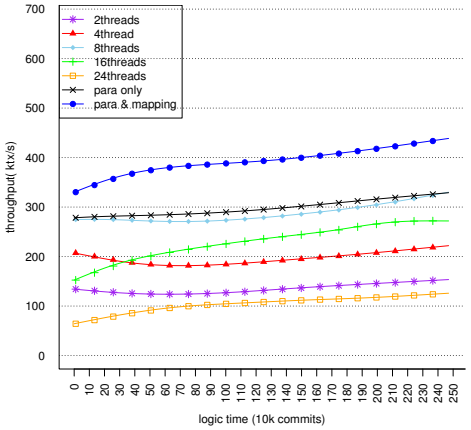
(b) **ssca2**



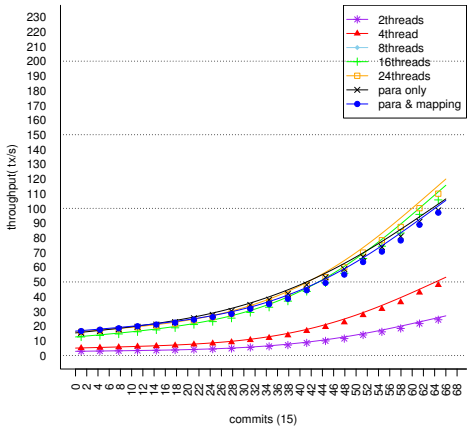
(c) **genome**



(d) **vacation**



(e) **yada**



(f) **labyrinth**

Figure 6.9: Time comparison of **STAMP** for static parallelism, *dynamic parallelism model* and *dynamic thread control model* on UMA. The dots represent the execution time with different static parallelism.

6.4.2 Results on the NUMA Machine

This section only gives the results on execution time comparison for the NUMA machine as indicated in Table 6.3 and Table 6.4. This section concentrates on the execution time comparison between the UMA and NUMA platforms. Since the execution time of **EigenBench** (two phases) rises drastically after 16 threads, the tables only include the time comparison for up to 16 threads for this benchmark. Overall, the performance of the dynamic models on the NUMA machine does not rival that on the UMA machine.

The *dynamic thread control model* gives **EigenBench** performance degradation against the best case on the NUMA, *per contra* it illustrates significant performance rise on the UMA machine. Switching the mapping strategies causes higher cost by thread migration on the NUMA platform. Furthermore, the throughput obtained from each strategy is less stable than that on the UMA platform, consequently the choice of the optimum strategy becomes inconstant.

vacation demonstrates better performance on the NUMA machine, as its control actions are taken less frequently than those on the UMA machine. Based on the observation of its program behaviour when the maximum parallelism degree is applied, its CR indicates some discrete abrupt changes. On the UMA machine, the frequency of such changes is higher than that on the NUMA machine. Therefore, more control actions that are mostly unnecessary are triggered on the UMA machine.

labyrinth shows similar behaviour on both machines due to its long transaction length and low contention. The maximum parallelism degree and the default thread mapping strategy are always predicted for this applications.

The NUMA machine also outperforms the UMA machine for **yada** with the dynamic models. **Compact** delivers the optimum performance for **yada** by mapping threads to the sibling cores. Mapping threads to the cores which share the same main memory can result in significant performance gain, since data accessing to the remote memory is more costly.

As explained in Section 4.5.3, **ssca2** suffers from the cost of calling the monitor. This cost is negligible and can be significantly reduced on the UMA machine, but it is significant on the NUMA machine. Therefore, **ssca2** indicates worse performance on the NUMA machine.

Lastly, **intruder** performs worse on the NUMA machine, as its controller predicts higher parallelism degree than the optimum during its first half execution. Although its parallelism degree is later modified to the near-optimum value, the performance penalty in the first half of its execution can hardly be offset. The reasons for such behaviours have been explained in Section 4.6.

benchmarks	best case	average value	worse case
EigenBench (two phases)	+4% (4)	+67%	+87% (16)
genome	-11% (4)	+91%	+99% (26)
vacation	-21% (8)	+90%	+96% (30)
labyrinth	+3% (32)	+52%	+87% (2)
yada	-5% (12)	+91%	+97% (28)
ssca2	-56% (6)	-47%	-41% (32)
intruder	-72% (6)	+46%	+68% (32)

Table 6.3: Performance comparison of different applications with the *dynamic parallelism model on NUMA*. The performance is compared with the minimum, average and the maximum value of all the static parallelism.

6.5 Discussion

The overhead of the approaches in this chapter mainly originates from three aspects: (1) Thread migration. This stems from two points: switching among thread mapping strategies and periodically wakening/suspending the threads to ensure execution time fairly distributed to each thread. (2) Unnecessary profiling of thread mapping strategies. Most of the transactions within **STAMP** applications show very similar behaviour and require no more profiling of mapping strategies despite of the parallelism fluctuations later. The more frequently the parallelism is adjusted, the higher overhead the

benchmarks	best case	average value	worse case
EigenBench (two phases)	-22% (4)	+59%	+83% (16)
genome	-18% (4)	+91%	+99% (26)
vacation	-12% (8)	+91%	+96% (30)
labyrinth	+3% (32)	+52%	+87% (2)
yada	+32% (12)	+94%	+98% (28)
ssca2	-56% (6)	-48%	-41% (32)
intruder	-27% (6)	+60%	+76% (32)

Table 6.4: Performance comparison of different applications with the *dynamic thread control model* on NUMA. The performance is compared with the minimum, average and the maximum value of all the static parallelism

applications suffer. (3) The cost of calling the monitor. Two factors contribute to that: the operation of obtaining and releasing the monitor and the time spent contending for the monitor. The latter cost rises significantly with the increment of active threads and gives significant impacts on the applications with short-length transactions. The cost of calling the monitor is negligible on the UMA machine and high on the NUMA machine as explained in Section 3.3.2.

The program phase change is determined by the advanced phase detection algorithm in this chapter. The limitations of the CR decision function become obvious on **intruder**. Since its CR tends to fluctuate frequently over the CR range, yet remains in the same phase. The *dynamic parallelism model*, therefore, overreacts to such changes and gives unstable performance. With better controls of thread migration, the *dynamic thread control model* is more resilient against abrupt parallelism changes, thus performs better. It is worth noting that Fig. 6.7(a) only presents the best case where the parallelism is adjusted less frequently.

The parallelism predictor relies on two assumptions which are based on ideal situations, thus the predicted parallelism *de facto* may be sub-optimum (*e.g.* **yada**). However, such performance loss is compensated by the performance gain from thread mapping. The gain obtained with a dynamic approach over a static one is directly related to the diversity of application phases. The more distinct the phases are, the higher performance gain it can obtain. The proposed dynamic approaches are hence more interesting to the applications with online behaviour variations. Although the dynamic framework is capable of optimising the parallelism and thread mapping strategy for applications with stable online behaviour, the performance benefit can not always compensate the profile overhead, as can be observed from **ssca2**. It is also worth noting

that not all the applications require to profile a thread mapping strategy. For instance, applications with low contention and their parallelism degrees equalling the core number (*e.g.* **labyrinth**, **ssca2**). Both models illustrate similar performance on the two applications. Additionally, profiling the thread mapping strategy penalises **genome** and **vacation**, since the two applications contain sudden contention changes which the *dynamic parallelism model* can respond to immediately. On the contrary, the *dynamic thread control model* requires extra profile lengths to search a better thread mapping strategy, meanwhile the applications have already transitioned to a new phase.

Compact is favoured when CR is low and **Scatter** is likely to be selected when CR is high. When CR is high, the *dynamic thread control model* favours high parallelism. When the parallelism is approaching the maximum core number, the thread mapping strategy profiling is disabled as little performance impact can be received from mapping threads. **Scatter** and **Compact** are based on the cache share. The two strategies do not take the sharing of the main memory into account, therefore, the thread mapping strategy does not perform as well as on the UMA machine.

It is worth noting the advanced phase detection algorithm demonstrates a significant performance rise against the simple phase detection algorithm (Chapter 4) on the NUMA platform for **ssca2**. The simple phase detection algorithm tends to overreact to the CR fluctuation. Continuous computation of CR range is performed by the controller, hence the thread number is frequently increased and decreased by one to obtain the CR range. In contrast, the advance phase detection algorithm prescribes a constant CR range during the complete execution of **ssca2**, resulting in a constant parallelism degree during its entire execution.

On the NUMA machine, the cost of thread migration is higher than that on the UMA machine. The performance on NUMA fluctuates more than that on UMA especially when both parallelism degree and thread mapping strategy are adapted at runtime. Therefore an application, which manifests variation of both parallelism and thread mapping strategies, tends to be less stable (*e.g.* **EigenBench**). Overall, the dynamic models indicate their limitations on performance improvement for the NUMA machine, that is the gain from thread control can hardly be offset by its overhead and the more complex memory structure of NUMA impacts on the optimum parallelism prediction. A better solution in the future work would be to reduce the initial parallelism degree and restrict the threads to a local memory bank so that optimum parallelism prediction can be less affected by thread migration.

6.6 Conclusion Remarks

This chapter has presented autonomic adaptation of parallelism and thread mapping on TinySTM. It has investigated the complexity of adjusting both parameters at runtime and proposed the solutions. The approaches are detailed next and their performance is compared with static parallelism. A feedback control loop is employed to coordinate the thread parallelism and mapping at runtime. Lastly, the implementation overhead has been analysed and the advantages as well as limitations of the work have been discussed.

Runtime parallelism adaptation can reduce contention among threads thus improve application performance. However, this does not consider the underlying hardware architecture. Since access latency rises from the low-level cache to the main memory, thread placement on the core can impact on application resource utilisation. Furthermore, adapting thread online causes thread migration, which leads to performance loss. The two issues mentioned above that cause performance degradation can be ameliorated by mapping threads to specific cores. However, dynamically controlling both parallelism and thread mapping is non-trivial. The proposed approach in this chapter on thread control demonstrates performance improvements on some applications against the static best case on the UMA machine. On the NUMA machine, however, the performance improvement is less promising, as the mapping strategies ignore data share among the distributed memory. New thread mapping strategies that take distributed memory share into account can better support the NUMA structure. It can reduce the influence of data access to remote memory on parallelism prediction by diminishing the starting thread number and restricting them to a local memory bank.

Four thread mapping strategies are profiled (three on the NUMA machine) in order to determine the optimum one. Such a profiling procedure is costly, as it induces thread migration and also forces the program to work partly under an unsuitable mapping strategy. A better approach for mapping prediction would be to utilise a predictor that can predict the optimum strategy in one step rather than four (three on NUMA) steps.

Chapter 7

Related Work

Chapter 7

Related Work

This chapter reviews prior research related to the work of this thesis. Literature on adaptation of thread parallelism and mapping shows a variety of approaches. These methods are investigated in three aspects as how the methodologies of the thesis are presented in the previous three chapters. Section 7.1 briefly describes the work on dynamic optimum parallelism detection with or without control techniques for TM systems. The following section (Section 7.2) reviews the work on thread mapping adaptation, but not limited to TM systems. Lastly, Section 7.3 introduces the state-of-art work on coordination of thread parallelism and mapping adaptation.

7.1 Dynamic Parallelism Adaptation on TM systems

It has been addressed in previous work [74, 75, 39, 76, 77, 78, 79, 41] to dynamically adapt thread parallelism via control techniques to reduce wasted work for TM systems.

Ansari et al.[74, 75] first proposed to adapt parallelism online by detecting the changes of the application's CR (commit ratio, see Section 2.3.1 for definition) using control techniques. The control action is made to parallelism if the CR falls out of the pre-set CR range [75] or does not equal a single pre-fixed CR value [74]. This is based on the fact that the CR falls during highly-contended phases and rises with low-contended phases. *Ansari et al.* gave five different algorithms which decide the sampling interval and the level of parallelism based on the CR. The first algorithm, called *SimpleAdjust*, increases or decreases threads by one when current CR is above or below the CR thresholds. *ExponentialInterval* extends *SimpleAdjust* by halving the sampling interval when the thread number has been changed or doubling it when the parallelism has not been varied. *ExponentialAdjust* keeps the sampling interval

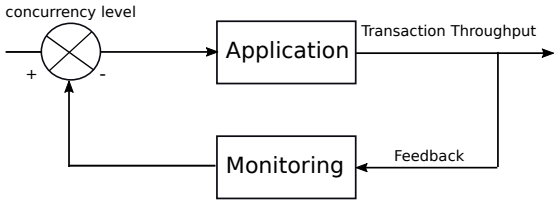


Figure 7.1: Flux Concurrency Control in a feedback-driven loop

fixed but exponentially increments or diminishes the thread number for every 10% of CR that falls out the CR range. *ExponentialCombined* combines *ExponentialInterval* and *ExponentialAdjust*, which improves the responsiveness of adaptation to the CR change. Additionally, a P-only algorithm is given, which determines the thread number to manipulate by sensing the difference between the current CR and a pre-set CR value. The number of threads to change is the current value multiplied by the CR difference and divided by 100.

Ravichandran et al. [76] presented a model which employs a feedback control loop with throughput (see Section 2.3.1 for definition) as the performance metric to adapt concurrency levels. The concurrency level is adapted in two phases, *i.e.* exponential and linear in the loop as shown in Fig. 7.1. The thread number is varied exponentially giving a better throughput while linearly giving a worse throughput. The concurrency level is halved in the first phase before switching to the second phase. During the second phase, the concurrency level increases giving a better throughput and decreases giving a worse throughput.

Rughetti et al. [39] utilise a neural network to enable performance prediction of STM applications. The neural network is trained to predict the execution time of wasted transactions which in turn is utilised by a control algorithm to regulate parallelism. The statistics required by the neural network are the average read-set size (rs_{size}), the average write-set size (ws_{size}), the average execution time for committed transactions (t_{time}) and the average execution time for the code block where the transactions and non-transactions interleave (ntc_{time}). In addition, two indices rw_{aff} and ww_{aff} are calculated which indicate the conflict affinity of read-write operations and write-write operations. The above statistics and indices together with the current concurrency level (k) are used to estimate wasted execution time as shown in Equation 7.1. The system structure is illustrated in Fig. 7.2.

$$w_{time} = f(rs_{size}, ws_{size}, rw_{aff}, ww_{aff}, t_{time}, ntc_{time}, k) \tag{7.1}$$

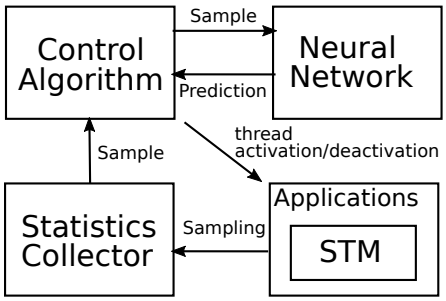


Figure 7.2: System architecture of *Rughetti et al.*' feedback control loop [39].

The optimal concurrency level is the value when the result of Equation 7.2 is maximised. Where i is the optimal concurrency level.

$$\frac{i}{w_{time,i} + t_{time,i}, ntc_{time}} \tag{7.2}$$

Rughetti et al. proposed an additional approach in [40], which depends on machine-learning to regulate concurrency level at runtime. This paper shares the same method as in [39] to predict the optimal parallelism. However, in preference to prediction based on the same amount of statistics, the size of the input features are shrunk or enlarged according to the quality of the estimated wasted time (contrasting with the real wasted time observed at runtime). More sampling overhead is incurred when more features are selected for prediction, consequently, input feature set is enlarged when the quality of wasted time is low while shrunk when it is high.

Similar to the two foregoing works from *Rughetti et al.*, *Sanzo et al.* [41] utilise the identical parameters and function as in Equation 7.1 to estimate the transaction abort probability rather than the wasted time. The transaction abort probability is expressed in Equation 7.3.

$$p_a = 1 - e^{-\rho.\omega.\phi} \tag{7.3}$$

Where p_a is the transaction abort probability; ρ is a function presumably dependent on input parameters $rs_{size}, ws_{size}, rw_{aff}$ and ww_{aff}; supposedly, the function ω depends on the parameter k; function ϕ is assumed to rely on the parameters t_{time} and ntc_{time}. ρ, ω and ϕ are derived by varying the workload, which is achieved by continuously changing certain aforementioned parameters while keeping the rest constant. Lastly, the optimal concurrency level is estimated by Equation 7.2. The derivation of all the parameters for the two equations is done through regression analysis that is obtained

by exploiting a set of sampling data gathered via runtime observation of applications. The approach of concurrency regulation proposed by *Sanzo et al.* is limited to the optimistic concurrency control where transactions are aborted and resumed right upon conflicts.

Didona et al. [77] described two approaches to identify optimal parallelism for shared-memory STM as well as distributed STM. The exploration-based approach is applied for shared-memory STM by trying different thread number to maximise the throughput. However, since the cost for trying different parallelism degrees is high for distributed STM, the authors combine a model-driven approach and the exploration-based approach to estimate optimal parallelism degree. This hybrid approach exploits a machine-learning method to build a function utilised for parallelism prediction. To deduce the function, the system collects the following features (which are called workload): duration and relative frequency, of read-only and update transactions, abort rate and average number of writes per transaction. By trying different parallelism degrees, the prediction function can be continuously corrected.

Wamhoff et al. [79] presented a synchronisation strategy called FASTLANE for STM systems that determines optimal synchronisation strategy for applications apropos of instrumentation code by detecting the active thread number. Nevertheless, the parallelism degree is determined at runtime. The authors argued that the performance of a single-threaded application without instrumentation is generally higher than that of STM systems with a small number of threads [80, 81, 57]. Furthermore, the paper stated that STM, with compiler instrumentation for transactions, requires over four parallelism degrees on average in order to outperform sequential code [82]. This work proposed two types of threads: one *master thread* that always executes without being aborted and the other *helper threads* help the master thread. The master thread is light while the helpers include more instrumentation code and thus slower. The paper proposed four code paths for each transaction: (1) a light sequential path without instrumentation, (2) a pessimistic path with a master thread lightly instrumented with writes, (3) a speculative path with helper threads that are instrumented for reads and deferred writes , (4) a full STM path with instrumented reads and writes. One code path can be selected dynamically at the beginning of a transaction, which enables applications to execute sequentially or in parallel.

The methodologies of dynamic parallelism regulation above are applied to STM. These methods are generally difficult to apply to HTM systems due to their heavy instrumentation. Studies (*e.g.* [78]) targeting on HTM have depicted their prominence

in parallelism adaptation.

Rughetti et al. [78] argued that the existing techniques for parallelism regulation on STM may not function effectively when applied to HTM. The paper presented two automatic approaches of tuning parallelism degrees on HTM. The two approaches require Decision Trees and Neural Network to obtain the relation between optimal parallelism and workload. The workload consists of the average execution time for committed transactions (t_{time}) and the average execution time for the code block where the transactions and non-transactions interleave (ntc_{time}), abort rate due to data conflict ($abort_{conflict}$), abort rate due to overflow of cache capacity ($abort_{capacity}$) and abort rate due to other architecture reasons ($abort_{other}$). The above two machine-learning based approaches make use of training data.

Comparing with the work of *Ansari et al.* [75, 74], this thesis resolves a CR range (used for phase detection) which is adaptive to the online program behaviour rather than a fixed range or a single prefixed value. This makes parallelism adaptation more sensitive to application phase change. Analogising with the parallelism prediction by *Ravichandran et al.* [76], *Rughetti et al.* and *Didona et al.* [77], the probabilistic model presented in the thesis predicts the optimum parallelism based on probability theory, which requires no offline training procedure or tries of different parallelism degree to search the optimum. This thesis is concerned about applications running with no less than two threads since transactional memory is designed for parallel applications. In contrast, *Wamhoff et al.*'s work [79] enables applications execute in sequential or parallel. Its selection of parallelism degree depends on the available cores meaning that the parallelism degree is not adaptive to application online behaviour fluctuations. *Wamhoff et al.* favour choices of the optimal synchronisation strategy when the parallelism degree is determined rather than adjusting it dynamically. Contrasting with the aforementioned works which either use CR or throughput to indicate performance, the thesis employs both: CR to indicate program phases and throughout to indicate correctness of parallelism regulation. Since either by itself is not sufficient enough to represent program performance (recall Section. 3.2.1). Furthermore, the thesis explicates the design of controllers as automata that is widely employed to describe and construct the controllers in the control theory. Automata facilitate the description of relations of all the control actions and explicitly illustrate their control frequency. Such designs, to the best of the author's knowledge, have not been found from the aforementioned studies.

7.2 Thread Mapping Adaptation

Literature has shown interesting work [18, 83, 84, 85, 86] which address the thread or process mapping issues on parallel applications. Although the previous work is not dedicated to TM, it presented relevance to the work of the thesis.

Diener et al. [18] described two methods, *i.e. Exhaustive Search* and *Heuristic Algorithm*, to resolve thread placement issues. The authors proposed a data sharing metric used to measure how much a certain thread placement can benefit from data sharing. This is calculated by aggregating the shared cache access between two threads with a high metric indicating high data sharing among caches. The exhaustive search method, as its name indicates, utilises exhaustive search via trying every possible thread placement on the cores, and selects the optimal placement for the one indicating the highest data sharing metric. This method is only feasible for a small number of threads due to the high cost. The heuristic algorithm, although it does not detect the best thread placement, requires less simulation time. Since it only searches the amount of data sharing for all the possible pairs of threads, meaning that certain groups of thread placement are skipped. Both methods require to perform simulation of the applications in order to obtain data sharing metrics.

Zhang et al. [83] presented a process mapping strategy for MPI (Message Passing Interface) [87] applications with collective communications (collective communication requires synchronisation of all the processes in a group). The collective communications are decomposed into a series of point-to-point communications. The authors then employ an existing strategy, *i.e.* the graph partitioning algorithm, to generate an appropriate process mapping for applications.

Jeannot et al. [84] proposed an algorithm called TREEMATCH that maps processes of MPI applications to computing elements based on the hierarchy topology of the target environment and the communication patterns of the different processes. The algorithm firstly constructs all the possible combination of the processes. It then traces the point-to-point and collective communications among all the groups and selects the one which delivers the highest degree of decline in communication cost.

Hong et al. [85] depicted a dynamic thread mapping scheme that maps threads to diverse processors according to workload of threads and frequency of processors. The mapping scheme is performed in two phases. In the first phase, called *detection phase*, an application is executed for one iteration of the loop nest of the thread. Processor cycles and cache access are recorded accordingly. The threads are remapped based on its workload. The thread with the most load is assigned to the processor-cache pair

with the combination of the fastest frequency and cache latency. The second phase starts by running the threads with the new mapping and it also records the workload information. No remapping is performed if the workload does not vary after a certain period, otherwise a new mapping is scheduled.

Tournavitis et al [86] stated a machine-learning method to choose the best mapping strategy from the four options provided by OpenMP [88], which are *CYCLIC*, *DYNAMIC*, *GUIDED* and *STATIC*. *Support Vector Machine (SVM)* is used to obtain classification indicating the mapping strategy to schedule. However the proposed mapping strategies target on how to schedule jobs to threads rather than scheduling threads to the specific CPU core.

Comparing with the aforementioned studies with respect to thread mapping on OpenMP or MPI systems for regular parallel programs, this thesis is concerned with TM applications for STM platforms. A TM system differentiates rationales of synchronisation from lock-based systems. A change of design policies (*e.g.* contention manager) can lead to drastic performance fluctuations for threads, thus complicating designs of thread mapping strategies.

Castro et al. [89, 90, 5] carried out the first study on thread mapping for TM systems. Their work have defined three novel thread mapping strategies (**compact**, **scatter**, **RoundRobin**, see Section 2.1.1 for more details) based on the sharing of diverse cache levels, together with **Linux** (the default mapping strategy), to leverage resource usage. The proposed approach incorporates an offline data training phase and an online profiling phase. A machine learning method is performed in the first phase to construct a function which predicts the optimal thread mapping strategy. The machine learning procedure takes into account of the following features: transaction time ratio (tx time/execution time), transaction abort ratio, conflict detection policy (eager and lazy), conflict resolution policy (backoff and suicide), cache miss ratio as well as the four aforementioned thread mapping strategies. A decision tree classifies the features and yields the optimal thread mapping strategy as the tree root. In the second phase, applications are periodically profiled so that the system can adapt the thread mapping strategy based on their online behaviour. The thread mapping approach presented in this thesis is a continuation of their work. In preference to machine-learning based prediction, each strategy is profiled and the one that achieves the highest throughput is applied as the optimal strategy. The thread mapping profiling procedure is implemented in a feedback control loop.

7.3 Coordination of Parallelism and Thread Mapping Adaptation

The literature indicates no prior work addressing the issue on coordination of thread parallelism and mapping for TM systems. Some previous studies [91, 92] investigated the decision of parallelism degrees and processor allocation for non-TM parallel applications. However, neither addresses the issue on pinning threads to the specific cores.

Wang et al. [91] developed a compiler-based, automatic approach to decide parallelism and scheduling strategies for OpenMP programs. An offline data training is performed to construct predictors. Two machine learning algorithms are utilised: a *feed-forward Artificial Neural Network* (ANN) to decide parallelism degrees and a *Support Vector Machine* (SVM) to determine thread scheduling rules (*i.e.* how tasks are scheduled to certain threads). Executing a program with its least amount of input data, the predictors are able to deduce the optimal thread configuration. In comparison with this work, this dissertation employs online profiling with the thread configurations varying based on the in-time fluctuations of program workload. Runtime adaptation benefits applications with runtime behaviour diversity. Furthermore, no offline training is required for prediction in the approaches presented in this thesis.

Corbalan et al. [92] proposed an approach that can select the number of threads and allocate tasks to the available threads. This approach is only concerned with the number of available processors that can be allocated for an application. Comparing with this work, this dissertation exploits hardware hierarchies at runtime and pins threads to specific cores. This can take advantage of access latency difference between levels of memory.

This thesis does not address the issue of task assignment to the available threads, rather it is more interested in threads online interaction. The proposed methodologies are concerned with reducing the global contention and improving the resource utilisation by controlling parallelism degrees and thread mapping. However, it is less interesting for OpenMP applications to consider issues on contention.

7.4 Conclusion Remarks

This chapter has reviewed some related works of significance and compared them with the work of this dissertation. The current research offers insights into runtime parallelism degree adaptation using control techniques. The feedback control loops are

either explicitly or implicitly stated and utilised in the system to regulate parallelism degrees. The parallelism degree can be obtained by trying its different values based on the application contention or throughput. It can also be determined by a machine-learning algorithm or model-driven algorithm being served as runtime prediction. The machine learning algorithm requires offline data training and the model-driven algorithm requires a large set of sampling data. The thread mapping strategy can give a significant impact on system performance. Literature on thread mapping focused on conventional parallel applications for OpenMP or MPI platforms with one notable exception by *Castro et al.* [89, 90] who coped with thread mapping strategies on STM systems. *Castro et al.* exploits a machine-learning algorithm to predict and adjust the thread mapping strategies according to the program behaviour. Some previous studies synthesised prediction of parallelism and thread mapping, yet at compile time and on conventional parallel applications.

Chapter 8

Conclusion and Future Work

Chapter 8

Conclusion and Future Work

Multi-core processors can enhance application performance by providing many cores to support application-level parallelism. A high parallelism degree can decrease computing time, however, potentially increases synchronisation cost. Therefore, parallel applications need to handle the trade-off between synchronisation and computation. The conventional approach to address synchronisation is to implement locks. The pitfall of this approach is the risk of deadlocks. Furthermore, it is intricate to analyse interaction among concurrent operations. Transactional memory provides an alternative way for writing parallel applications. Rather than utilising locks to block concurrent access, transactional memory offers programmers interfaces to enclose concurrent access in transactions. Transactions execute speculatively without being blocked by locks. Transactional memory provides a lock-free environment for programmers.

8.1 Conclusion

This thesis presents work on autonomic management of thread parallelism and mapping on TinySTM. The proposed methodologies address the issues arising from how to identify the trade-off between synchronisation and computation among multiple threads and how to reduce thread memory access latency. A suitable parallelism degree can speed up computation and keep synchronisation cost low. A good thread mapping strategy can reduce data access latency of threads. Although it is possible to analyse applications offline and make a tentative decision on the parallelism degree and thread mapping strategy, such a decision can only be applied to applications with stable behaviour. It requires a lot efforts to determine the settings for the applications with online behaviour variation from an offline view. Therefore, online application

analysis and decision making become necessary. Autonomic computing provides technical supports for automatic system management. The idea of autonomic computing is through implementing feedback control loops to monitor and continuously improve system performance. The feedback control loop keeps collecting feedbacks (impacts of its previous decision) from the system it manages, so that its decision on performance enhancement can be continuously improved.

This thesis presents the designs of feedback control loops that can dictate parallelism degrees and thread mapping strategies based on application runtime behaviours as well as the hardware architecture. Although literature has demonstrated insights into designs of feedback control loop to manage parallelism degrees for TM applications, no previous studies have been conducted on online manipulation of both. It is challenging to control both simultaneously for TM systems, as either can impact on prediction of the other. Moreover, controlling parallelism and thread mapping at runtime can easily lead to thread migration from one core to another, causing application performance degradation. This dissertation addresses the above two issues and illustrates the efficiency of the solutions by comparing runtime application performance of the proposed models with that of static parallelism degrees.

The frequency of control actions for a feedback control loop can significantly impact on its performance. The previous literature on TM reveals some attempts to address this issue either explicitly or implicitly, however, their decisions on frequency of control actions incorporate strong empirical bases (*e.g.* no parallelism adaptation when CR is within 30 % and 60%, or simply trigger the control action periodically). In industry automation control, control actions are often triggered by sensors which are based on elaborate mathematical functions. However such modelisations have not been extensively studied in computing systems. This thesis proposes the phase detection algorithms that can sense application phase changes, and the control actions are taken when a phase change is detected. Consequently, profiling overhead is reduced.

Despite the significance of the proposed methodologies, some limitations of this work also need to draw attention. The probabilistic model can predict optimum parallelism degree for an application at runtime. Its prediction requires no offline profiling to construct a predictor that is normally based on machine-learning approaches. The machine-learning based parallelism predictor tends to be platform-dependent. In contrast, the proposed parallelism predictor in the thesis is platform-independent, yet its derivation is based on ideal situations. Hence, the model can yield a sub-optimum parallelism degree *de facto*. Furthermore, regarding the decision of optimum thread

mapping strategy, it is not always guaranteed to obtain the same optimum strategy by profiling each and comparing their throughputs, since the performance of certain strategies are similar. Additionally, throughputs are affected by thread migration caused by periodically wakening and suspending threads.

The proposed approaches take profits of application runtime phase diversity to improve performance. The more distinct the phases are, the higher performance improvement the approaches can yield. Therefore, the performance benefits can surpass implementation overhead. Since the overhead is inevitable, it facilitates to diminish the overhead by analysis of its causes. The main overhead originates from:

1. Thread migration. Thread migration contributes a significant amount of overhead to the runtime adaptation approaches. Notably, the performance penalty caused by thread migration is significantly higher on the NUMA platform due to its non-uniform memory access.

2. Profiling the thread mapping strategies. It can be costly to try each strategy in order to determine the best option for an application, since the application works under the non-optimum strategies during profiling. Such cost is higher on the NUMA machine than that on the UMA machine.

3. Cost of calling the monitor. This cost is trivial on the UMA machine and it can be further reduced by trimming the calling frequency. However it is significant on the NUMA machine for the application with very short transaction length.

4. The parallelism degree to start with. The probabilistic model starts with the maximum core number (more specifically, the core number of the UMA machine), whereas the simple model begins with the minimum value. The former model requires a large amount of transactions executed by many threads at a fixed period to guarantee the constant probability of a conflict. A large number of threads can cause high contention. The simple model avoids excessive contention in the beginning but hinders progress of some applications meanwhile.

The impact of the listed overhead and limitations of the proposed approaches vary from diverse TM applications to different mechanisms of TM platforms as well as hardware architecture. This dissertation does not intend to provide a versatile solution that can work under all the circumstances, yet it offers approaches that can be applied for a wide range of applications and their supporting platforms.

Multi-core processors are increasing their number of cores and complexity of their memory hierarchy leading to higher requirements of parallel applications. A mechanism to simplify designs of parallel application becomes more and more essential. Meanwhile how to reduce synchronisation time and memory access latency for running applications becomes critical for system performance. The diversity of applications and underlying platforms (software or/and hardware) urges a design of a system that is capable of managing itself given the objectives from high-level administrators. This thesis illustrates significance and efficiency of the methodologies on runtime thread parallelism and mapping control using autonomic computing techniques for software transactional memory. It offers insights into designs of autonomic STM systems that can be otherwise adapted to other parallel programming platforms.

8.2 Future Work

The scale of the proposed methodologies in this thesis can be extended to a larger area with the aims to accelerate performance of parallel applications automatically. Exploration of the followings in future research can facilitate the attainment of this goal.

8.2.1 Thread Mapping Strategy

The thread mapping strategies in this thesis do not take into account of access latency diversity of distributed memory. One interesting topic for the further work can be carried out on designs of new thread mapping strategies that can diminish access latency apropos of remote main memory.

Furthermore, a thread mapping strategy predictor that can predict the optimum strategy after one profile length is achievable with possible assistance from compilers. Compilers are able to analyse and provide sufficient information (*e.g.* the memory-intensive and computation-intensive code) of applications as well their underlying hardware. This information together with runtime profile information can construct the predictor for the optimum mapping strategy.

8.2.2 From STM to HTM

STM is memory-consuming. In general, its performance gain from speculative transaction execution can hardly compensate its performance penalty from its significant

amount of memory log. The research trend on TM is seeking auxiliary from hardware. The idea of feedback control loops to facilitate designs of autonomic computing systems can be also applied to HTM. The sensors and actions of the feedback control loops stated in this thesis are not only specific to STM. Therefore, adapting the approaches from STM to HTM becomes possible provided that peculiarities of HTM system is taken into account. A salient peculiarity on HTM is its causes of aborts due to capacity constraints of processors (*e.g.* the capacity of cache line). Such a factor is absent from STM. In addition, the overhead from profiling the same parameters in the context of HTM can easily shield the performance enhancement from hardware. Further studies targeting HTM can utilise the methodologies in this thesis and yet need to address the aforementioned issues.

8.2.3 Coordination of Feedback Control Loops

This thesis utilises both periodical and event-based control to trigger control actions. Event-based control demonstrates its advantage in diminishing its control frequency thus causing less overhead for control systems. The control theory has developed numerous models, simple or sophisticated, to handle event-based control, *e.g.* PID controller [93]. As discussed in Chapter 6, a controller that can respond efficiently to program phase changes, meanwhile, reduce overshooting is feasible.

Performance influence on TM systems stems from multiple aspects. Future work can be conducted on multiple-objectives optimisation. Multiple-objective optimisation has been extensively studied in control theory. It can be addressed in one or several loops. Several control loops can either be controlled by one master loop, or are at equivalent position. When more than one feedback control loops exist, coordination of the loops becomes necessary. Alas, it is onerous to program multiple loops which incorporate complicated coordination using C/C++ or JAVA. The Heptagon/BZR programming language [94, 95] designed by INRIA provides a straightforward way to program automata and their coordination. This language can be utilised in further work to facilitate complex designs of loops for parallel platforms.

8.2.4 From STM to Other Parallel Platforms

The algorithms on parallelism prediction presented in this thesis may be adapted to other parallel platforms (*e.g.* OpenMP [88], Charm++ [96]) by substituting both CR and throughput with memory contention and instruction throughput.

A STM system is a high-level platform utilised to address synchronisation. The control system proposed in this thesis works on top of the STM system, therefore it provides more accessible interfaces to programmers than its managed element. The idea of utilising feedback control loops to automatically manage thread parallelism degrees and mapping strategies can be otherwise implemented in other high-level parallel platforms.

Bibliography

[1] J. Larus and C. Kozyrakis, "Transactional memory," *Commun. ACM*, vol. 51, pp. 80–88, July 2008.

[2] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, pp. 289–300, May 1993.

[3] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, (New York, NY, USA), pp. 237–246, ACM, 2008.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," *SIGPLAN Not.*, vol. 41, pp. 336–346, Oct. 2006.

[5] M. B. Castro, *Improving the Performance of Transactional Memory Applications on Multicores: A Machine Learning-based Approach*. PhD thesis, University de Grenoble, December 2012.

[6] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, Jan. 2003.

[7] N. Zhou, G. Delaval, B. Robu, É. Rutten, and J.-F. Méhaut, "Autonomic Parallelism and Thread Mapping Control on Software Transactional Memory," in *ICAC 2016 - 13th IEEE International Conference on Autonomic Computing* , (Wursburg, Germany), July 2016.

[8] N. Zhou, G. Delaval, B. Robu, É. Rutten, and J.-F. Méhaut, "Control of Autonomic Parallelism Adaptation on Software Transactional Memory," in *International Conference on High Performance Computing & Simulation (HPCS)*, (Innsbruck, Austria), July 2016.

[9] N. Zhou, G. Delaval, B. Robu, É. Rutten, and J.-F. Méhaut, "Autonomic Parallelism Adaptation for Software Transactional Memory," in *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS)*, (Lorient, France), July 2016.

[10] N. Zhou, G. Delaval, B. Robu, É. Rutten, and J.-F. Méhaut, "Autonomic Parallelism Adaptation on Software Transactional Memory," Research Report RR-8887, Univ. Grenoble Alpes ; INRIA Grenoble, Mar. 2016.

[11] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, Jan 1998.

[12] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[14] L. McVoy and C. Staelin, "Lmbench: Portable Tools for Performance Analysis," in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, (Berkeley, CA, USA), pp. 23–23, USENIX Association, 1996.

[15] A. Vajda, *Programming Many-Core Chips*. Springer Publishing Company, Incorporated, 1st ed., 2011.

[16] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, pp. 32–38, Nov. 2005.

[17] T. E. Anderson, D. D. Lazowska, and H. M. Levy, "The Performance Implications of Thread Management Alternatives for Shared-memory Multiprocessors," *SIGMETRICS Perform. Eval. Rev.*, vol. 17, pp. 49–60, Apr. 1989.

[18] M. Diener, F. Madruga, E. Rodrigues, M. Alves, J. Schneider, P. Navaux, and H.-U. Heiss, "Evaluating thread placement based on memory access patterns for multi-core processors," in *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pp. 491–496, Sept 2010.

[19] C. Pousa Ribeiro, M. Castro, V. Marangonzova-Martin, J.-F. Mehaut, H. C. D. Freitas, and C. A. P. D. Silva Martins, "Evaluating CPU and Memory Affinity for Numerical Scientific Multithreaded Benchmarks on Multi-cores," *IADIS International Journal on Computer Science and Information Systems (IJCSIS)*, 2012.

[20] A. Agawal and A. Gupta, "Memory-reference Characteristics of Multiprocessor Applications Under MACH," *SIGMETRICS Perform. Eval. Rev.*, vol. 16, pp. 215–225, May 1988.

[21] R. Thekkath and S. J. Eggers, "Impact of sharing-based thread placement on multithreaded architectures," in *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*, pp. 176–186, Apr 1994.

[22] J. A. Brown, L. Porter, and D. M. Tullsen, "Fast thread migration via cache working set prediction," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, (Washington, DC, USA), pp. 193–204, IEEE Computer Society, 2011.

[23] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and thread affinity in openmp programs," in *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?*, MAW '08, (New York, NY, USA), pp. 377–384, ACM, 2008.

[24] J. Antony, P. P. Janes, and A. P. Rendell, "Exploring thread and memory placement on numa architectures: Solaris and linux, ultrasparc/fireplane and opteron/hypertransport," in *Proceedings of the 13th International Conference on High Performance Computing*, HiPC'06, (Berlin, Heidelberg), pp. 338–352, Springer-Verlag, 2006.

[25] M. B. Greenwald, *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, 1999.

[26] S. Al Bahra, "Nonblocking algorithms and scalable multicore programming," *Commun. ACM*, vol. 56, pp. 50–61, July 2013.

[27] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, (New York, NY, USA), pp. 92–101, ACM, 2003.

[28] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 124–149, Jan. 1991.

[29] V. J. Marathe and M. L. Scott, "A Qualitative Survey of Modern Software Transactional Memory Systems," Tech. Rep. TR839, University of Rochester, June 2004.

[30] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi, "Optimizing memory transactions," *SIGPLAN Not.*, vol. 41, pp. 14–25, June 2006.

[31] D. B. Lomet, "Process structuring, synchronization, and recovery using atomic actions," *SIGOPS Oper. Syst. Rev.*, vol. 11, pp. 128–137, Mar. 1977.

[32] J. Gray, "Notes on Data Base Operating Systems," in *Operating Systems, An Advanced Course*, (London, UK, UK), pp. 393–481, Springer-Verlag, 1978.

[33] J. E. B. M. Maurice Herlihy, "Transactional memory: Architectural support for lock-free data structures," tech. rep., Digital Cambridge Research Lab, Digital Cambridge Research Lab, One Kendall Square, Cambridge MA 02139, December 1992.

[34] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2Nd Edition*. Morgan and Claypool Publishers, 2nd ed., 2010.

[35] M. Milovanović, R. Ferrer, O. S. Unsal, A. Cristal, X. Martorell, E. Ayguadé, J. Labarta, and M. Valero, "Transactional Memory and OpenMP," in *Proceedings of the 3rd International Workshop on OpenMP: A Practical Programming Model for the Multi-Core Era*, IWOMP '07, (Berlin, Heidelberg), pp. 37–53, Springer-Verlag, 2008.

[36] E. Vallejo, S. Sanyal, T. Harris, F. Vallejo, R. Beivide, O. Unsal, A. Cristal, and M. Valero, "Hybrid transactional memory with pessimistic concurrency control," *International Journal of Parallel Programming*, vol. 39, no. 3, pp. 375–396, 2010.

[37] Z. He, X. Yu, and B. Hong, "Profiling-based adaptive contention management for software transactional memory," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 1204–1215, May 2012.

[38] M. Ansari, K. Jarvis, C. Kotselidis, M. Lujan, C. Kirkham, and I. Watson, "Profiling transactional memory applications," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pp. 11–20, Feb 2009.

[39] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia, "Machine learning-based self-adjusting concurrency in software transactional memory systems," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pp. 278–285, Aug 2012.

[40] D. Rughetti, P. D. Sanzo, B. Ciciani, and F. Quaglia, "Dynamic feature selection for machine-learning based concurrency regulation in stm," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pp. 68–75, Feb 2014.

[41] P. D. Sanzo, F. D. Re, D. Rughetti, B. Ciciani, and F. Quaglia, "Regulating concurrency in software transactional memory: An effective model-based approach," in *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, pp. 31–40, Sept 2013.

[42] M. M. Pereira, M. Gaudet, J. N. Amaral, and G. Araujo, "Study of hardware transactional memory characteristics and serialization policies on haswell," *Parallel Computing*, pp. –, 2015.

[43] E. Koskinen and M. Herlihy, "Checkpoints and continuations instead of nested transactions," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, (New York, NY, USA), pp. 160–168, ACM, 2008.

[44] Intel Corporation, *Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extentions.*, 2012.

[45] N. Diegues and P. Romano, "Self-Tuning Intel Transactional Synchronization Extensions," in *11th International Conference on Autonomic Computing (ICAC 14)*, (Philadelphia, PA), pp. 209–219, USENIX Association, June 2014.

[46] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum, "Applications of the adaptive transactional memory test platform," in *the TRANSACT'08:3rd Workshop on Transactional Computing*, 2008.

[47] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, (New York, NY, USA), pp. 209–220, ACM, 2006.

[48] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, (Berlin, Heidelberg), pp. 194–208, Springer-Verlag, 2006.

[49] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," *SIGPLAN Not.*, vol. 44, pp. 155–165, June 2009.

[50] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, and M. L. Scott, "Lowering the Overhead of Nonblocking Software Transactional Memory," 2006.

[51] K. Fraser and T. Harris, "Concurrent programming without locks," *ACM Trans. Comput. Syst.*, vol. 25, May 2007.

[52] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. LujÃąn, and K. Jarvis, "Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory," in *Algorithms and Architectures for Parallel Processing* (A. Bourgeois and S. Zheng, eds.), vol. 5022 of *Lecture Notes in Computer Science*, pp. 196–207, Springer Berlin Heidelberg, 2008.

[53] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero, "RMS-TM: a comprehensive benchmark suite for transactional memory systems," *SIGSOFT Softw. Eng. Notes*, vol. 36, pp. 335–346, Sept. 2011.

[54] R. Guerraoui, M. Kapalka, and J. Vitek, "STMBench7: A Benchmark for Software Transactional Memory," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 315–324, Mar. 2007.

[55] F. Zyulkyarov, A. Cristal, S. Cvijic, E. Ayguade, M. Valero, O. Unsal, and T. Harris, "WormBench: A Configurable Workload for Evaluating Transactional Memory Systems," in *Proceedings of the 9th Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*, MEDEA '08, (New York, NY, USA), pp. 61–68, ACM, 2008.

[56] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "EigenBench: A simple exploration tool for orthogonal TM characteristics," in *2010 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–11, Dec 2010.

[57] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *2008 IEEE International Symposium on Workload Characterization (IISWC)*, September 2008.

[58] W. Ruan, Y. Liu, and M. Spear, "STAMP need not be considered harmful," in *9th ACM SIGPLAN Workshop on Transactional Computing*, (Salt Lake City), March 2014.

[59] C. Y. Lee, "An algorithm for path connections and its applications," *Electronic Computers, IRE Transactions on*, vol. EC-10, pp. 346–365, Sept 1961.

[60] J. Ruppert, "A delaunay refinement algorithm for quality 2-dimensional mesh generation," *J. Algorithms*, vol. 18, pp. 548–585, May 1995.

[61] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing — degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, pp. 7:1–7:28, Aug. 2008.

[62] IBM, "An architectural blueprint for autonomic computing," tech. rep., IBM, 2003.

[63] Y. Sun, J. Lifflander, and L. V. Kalé, "PICS: A Performance-Analysis-Based Introspective Control System to Steer Parallel Applications," 2014.

[64] I. J. Dooley, *Intelligent Runtime Tuning of Parallel Applications with Control Points*. PhD thesis, Champaign, IL, USA, 2010. AAI3455708.

[65] E. Rutten, N. Marchand, and D. Simon, "Feedback Control as MAPE-K loop in Autonomic Computing," in *Software Engineering for Self-Adaptive Systems*, Lecture Notes in Computer Science, Springer, Apr. 2016.

[66] M. Litoiu, M. Shaw, G. Tamura, N. M. Villegas, H. Müller, H. Giese, E. Rutten, and R. Rouvoy, "What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems?," in *Software Engineering for Self-Adaptive Systems 3: Assurances* (R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, eds.), Springer, Feb. 2016.

[67] H. Müller, M. Pezzè, and M. Shaw, "Visibility of control in adaptive systems," in *Proceedings of the 2Nd International Workshop on Ultra-large-scale Software-intensive Systems*, ULSSIS '08, (New York, NY, USA), pp. 23–26, ACM, 2008.

[68] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Software engineering for self-adaptive systems," ch. Engineering Self-Adaptive Systems Through Feedback Loops, pp. 48–70, Berlin, Heidelberg: Springer-Verlag, 2009.

[69] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[70] W. L. Brogan, *Modern Control Theory (3rd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991.

[71] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 2001.

[72] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, (Washington, DC, USA), pp. 217–, IEEE Computer Society, 2003.

[73] A. Silberschatz, *(WCS)Operating System Concepts 7th Edition Flex Format*. John Wiley & Sons, 2005.

[74] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "Advanced concurrency control for transactional memory using transaction commit rate," in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, (Berlin, Heidelberg), pp. 719–728, Springer-Verlag, 2008.

[75] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "Adaptive concurrency control for transactional memory," in *MULTIPROG '08: First Workshop on Programmability Issues for Multi-Core Computers*, January 2008.

[76] K. Ravichandran and S. Pande, "F2C2-STM: Flux-based feedback-driven concurrency control for STMs," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 927–938, May 2014.

[77] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory applications," in *Networked Systems* (V. Gramoli and R. Guerraoui, eds.), vol. 7853 of *Lecture Notes in Computer Science*, pp. 233–247, Springer Berlin Heidelberg, 2013.

[78] D. Rughetti, P. Romano, F. Quaglia, and B. Ciciani, *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, ch. Automatic Tuning of the Parallelism Degree in Hardware Transactional Memory, pp. 475–486. Cham: Springer International Publishing, 2014.

[79] J.-T. Wamhoff, C. Fetzer, P. Felber, E. Rivière, and G. Muller, "Fastlane: Improving performance of software transactional memory for low thread counts," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, (New York, NY, USA), pp. 113–122, ACM, 2013.

[80] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière, "Evaluation of amd's advanced synchronization facility within a complete transactional memory stack," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, (New York, NY, USA), pp. 27–40, ACM, 2010.

[81] L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear, *Transactional Mutex Locks*, pp. 2–13. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[82] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui, "Why stm can be more than a research toy," *Commun. ACM*, vol. 54, pp. 70–77, Apr. 2011.

[83] J. Zhang, J. Zhai, W. Chen, and W. Zheng, *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings*, ch. Process Mapping for MPI Collective Communications, pp. 81–92. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[84] E. Jeannot and G. Mercier, "Near-Optimal Placement of MPI processes on Hierarchical NUMA Architectures," in *Europar* (P. D'Ambra, M. R. Guarracino, and D. Talia, eds.), vol. 6272, (Ischia, Italy), pp. 199–210, Springer, Aug. 2010.

[85] S. Hong, S. Narayanan, M. Kandemir, and O. Ozturk, "Process variation aware thread mapping for chip multiprocessors," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pp. 821–826, April 2009.

[86] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping," *SIGPLAN Not.*, vol. 44, pp. 177–187, June 2009.

[87] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-passing Interface*. Cambridge, MA, USA: MIT Press, 1994.

[88] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Comput. Sci. Eng.*, vol. 5, pp. 46–55, Jan. 1998.

[89] M. Castro, L. F. W. G. Goes, and J.-F. Mehaut, "Adaptive thread mapping strategies for transactional memory applications," *Journal of Parallel and Distributed Computing*, vol. 74, no. 9, pp. 2845 – 2859, 2014.

[90] M. Castro, L. F. W. G. Goes, C. P. Ribeiro, M. Cole, M. Cintra, and J.-F. Mehaut, "A machine learning-based approach for thread mapping on transactional memory applications," in *High Performance Computing (HiPC), 2011 18th International Conference on*, pp. 1–10, Dec 2011.

[91] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," *SIGPLAN Not.*, vol. 44, pp. 75–84, Feb. 2009.

[92] J. Corbalán, X. Martorell, and J. Labarta, "Performance-driven processor allocation," in *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, (Berkeley, CA, USA), USENIX Association, 2000.

[93] K. H. Ang, G. Chong, and Y. Li, "PID control system analysis, design, and technology," *IEEE Transactions on Control Systems Technology*, vol. 13, pp. 559–576, July 2005.

[94] G. Delaval, N. De Palma, S. M.-K. Gueye, H. Marchand, and É. Rutten, "Discrete Control of Computing Systems Administration: A Programming Language Supported Approach," in *European Control Conference* (M. Morari, ed.), (Zurich, Switzerland), pp. 117–124, July 2013.

[95] G. Delaval, H. Marchand, and E. Rutten, "Contracts for Modular Discrete Controller Synthesis," in *ACM International Conference on Languages, Compilers,*

and Tools for Embedded Systems (LCTES 2010), (Stockholm, Sweden), Apr. 2010.

[96] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93* (A. Paepcke, ed.), pp. 91–108, ACM Press, September 1993.

Glossary

A | C | D | I | L | M | N | O | P | R | S | T | U | V | W

A

abort

A transaction has failed, its previous changes made to the data locations are abandoned. 14

actuator

It carries out changes to the managed element in a MAPE-K loop. 32

autonomic computing

Computing systems that can manage themselves given high level objectives from administrators. 31

autonomic manager

It is the controller of a MAPE-K loop. It takes the information of interest from the sensor to monitors and execute changes via effectors. 32

C

checkpoint

Is a program location within a transaction where control may jump during a partial abort. 20

commit

A transaction successfully finishes its operations and the changes to the memory are made permanently. 14

commit ratio

The number of commits divided by the number of commits and aborts. 19

compulsory miss

Due to the competition to the same cache from several threads which causes cache misses. 9

conflict detection

Decides when to check the read/write sets to detect conflicts. 16

contention manager

Decides the actions to be taken in order to resolve the conflicts. 17

D

dynamic thread mapping strategy

Means the thread mapping strategy is selected during execution and may vary during execution. 81

I

invalidation miss

When the data has already resided in the cache but is evicted by other cache lines. 9

L

lemming effect

One transaction aborts and acquire a lock to fall back to the lock-based execution, consequently causing other transactions to abort and endeavour to obtain the lock. 21

lock freedom

At least one process or thread is guaranteed to complete in a finite number of steps. 13

M

managed element

Can be any software or/and hardware that is given autonomic behaviour by coupling it with an autonomic manager in a MAPE-K loop. 33

MAPE-K loop

It is a feedback control loop proposed by IBM. MAPE-K stands for Monitor, Analyse, Plan, Execute, Knowledge. 32

memory affinity

It is ensured when data is efficiently distributed over the machine memory. 11

N

non-action interval

Is composed of one or a continuous sequence of profile lengths, within which the thread regulation is suspended. 41

non-blocking algorithm

When an algorithm guarantees that at least one process can complete a task or make progress within a finite time. 12

Non-Uniform Memory Access

Multiprocessors with distributed memory, which have different access to local and remote memory. 7

O

obstruction freedom

Provides single-thread progress guarantees in the absence of conflicting operations. 12

optimistic concurrency control

The conflicts detection and resolution can be delayed after data access. 15

P

pessimistic concurrency control

Detects and solves the conflicts at the same time when a transaction is about accessing a location. 15

profile length

Is a fixed period for information gathering, such as commits, aborts and time. 41

profiling

Profiling in software engineering refers to a form of dynamic application analysis that measures certain useful events to facilitate application optimisation. 40

R

readset

A readset of a transaction is the set of locations read by the transaction.. 18

S

sensor

It is also called probe or gauge, which collects information on the managed element in a MAPE-K loop. 32

static thread mapping strategy

Means that the thread mapping strategy is chosen before execution and remains constant at runtime. 81

T

thread affinity

Fixing a thread to a specific core is called setting the. 9

thread control

To simplify the description, the control of both thread parallelism and thread mapping is addressed as thread control. 91

thread mapping

Assigning multiple threads to specific cores on a multi-core platform. 9

thread profile interval

Is composed of a continuous sequence of profile lengths within which the parallelism or thread mapping strategy is adjusted and the CR range is computed. 41

thread synchronisation

Threads must communicate and exchange data to complete their tasks. 11

throughput

The number of commits in one unit of time. 19

transaction

Is a finite sequence of machine instructions, executed by a single process. 14

transaction granularity

Is the unit of storage in which a TM system detects conflicts. 16

transactional memory

An alternative parallel programming technique, which addresses synchronisation issues through transactions. The access to the shared data are enclosed in transactions which are executed speculatively without blocking by locks. 13

U

uniform memory access

All the processors have the equal access to one single centralized memory. 7

V

version management

Handles the storage policy for permanent and transient data copies. 17

W

wait freedom

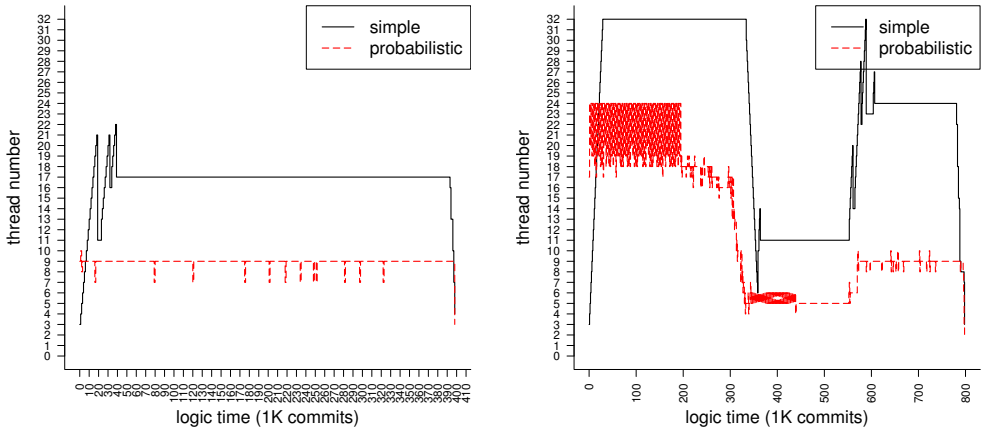
Every active process or thread can complete in a finite number of steps. 13

writeset

A writeset of a transaction is a set of locations accessed by the transaction. 19

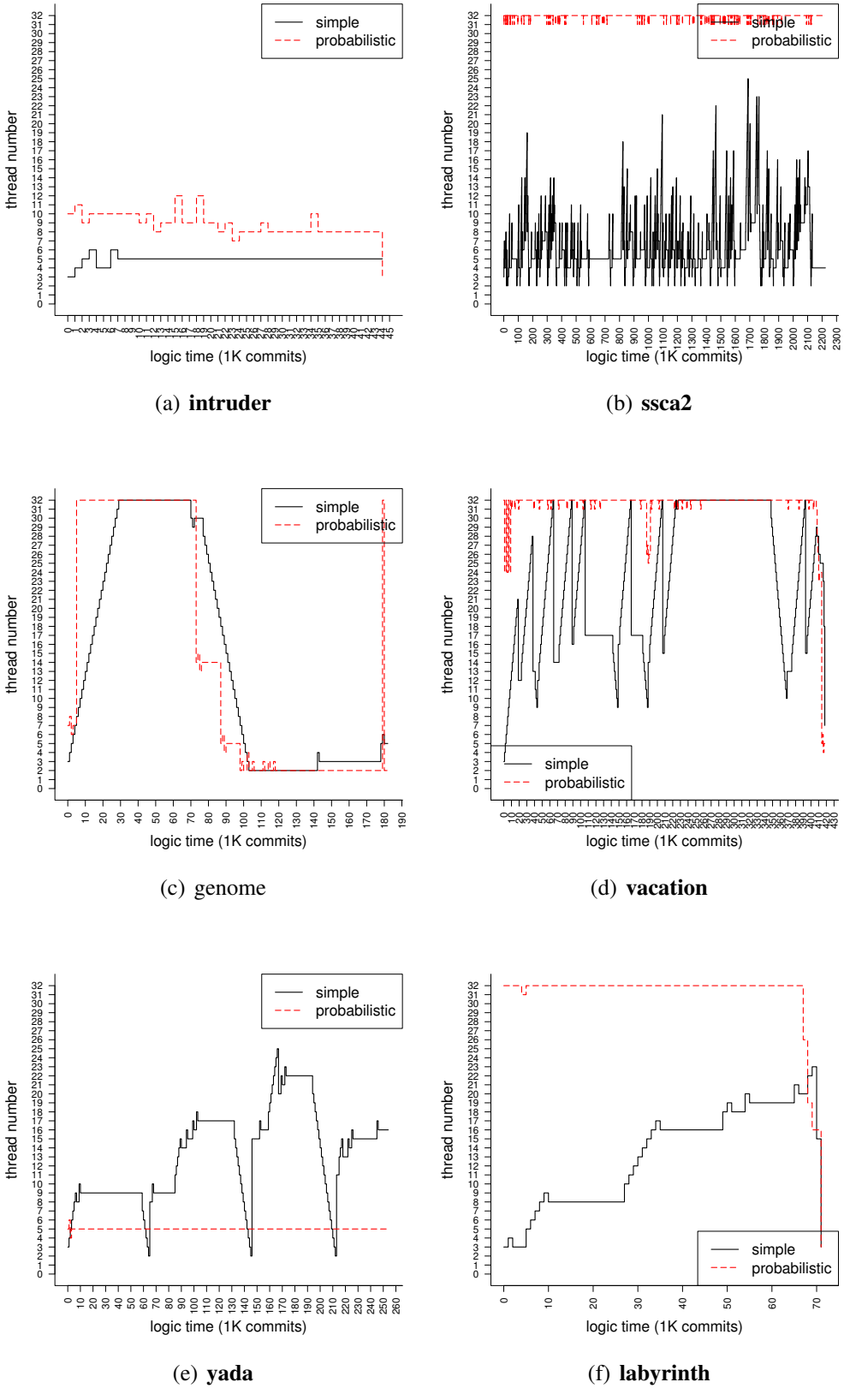
Appendix A

Runtime Parallelism Variation by the Simple and Probabilistic Models on NUMA



(a) one phase (b) three phases

Figure A.1: Runtime parallelism variation by the simple and probabilistic models for **EigenBench** on the NUMA platform.



(a) **intruder** (b) **ssca2**

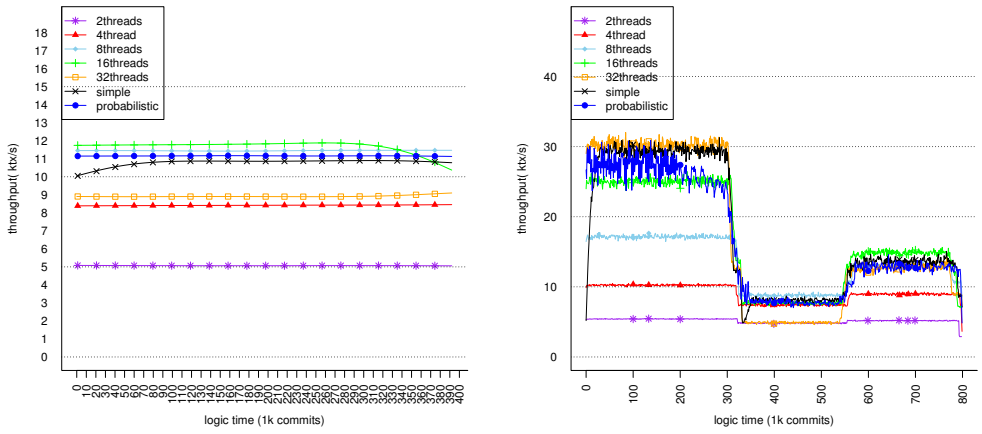
(c) genome (d) **vacation**

(e) **yada** (f) **labyrinth**

Figure A.2: Runtime parallelism variation by the simple and probabilistic models for **STAMP** on the NUMA platform.

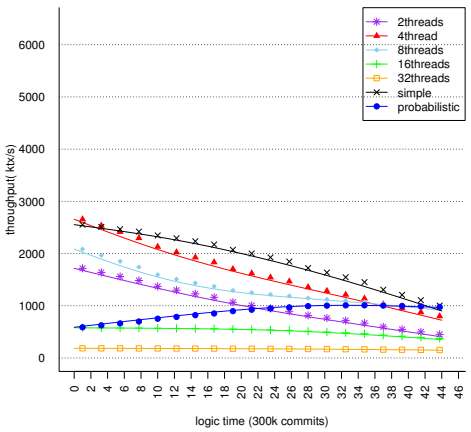
Appendix B

Runtime Throughput Comparison on Static Parallelism and Dynamic Parallelism by the Simple and Probabilistic Models

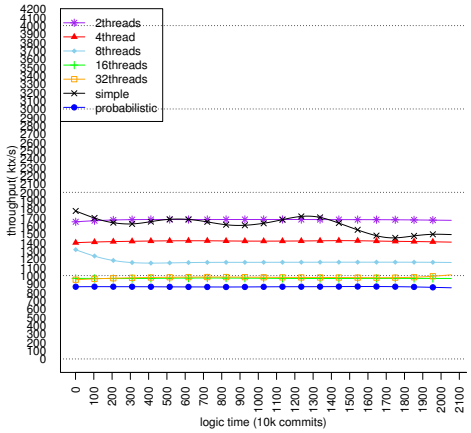


(a) one phase (b) three phases

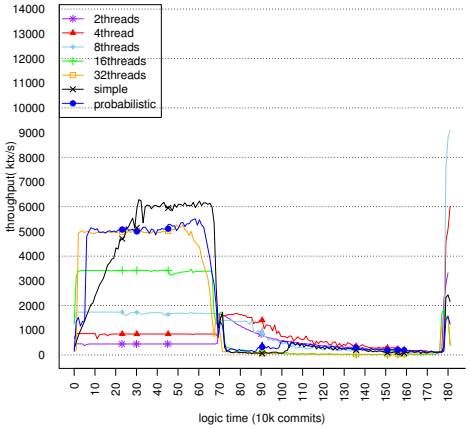
Figure B.1: Throughput comparison for **EigenBench**. Fig. B.1(a) presents the regression curve of the throughput over time, meaning that the throughput has been smoothed. Fig. B.1(b) illustrates the original data to better present the clear phase change.



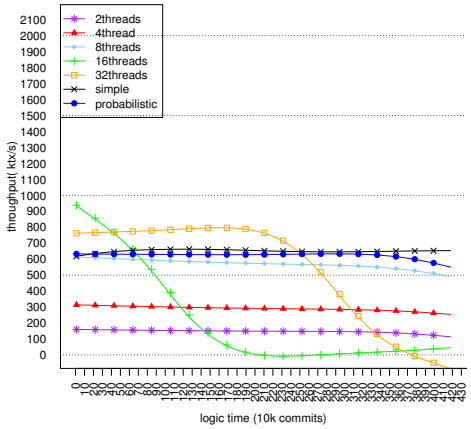
(a) **intruder**



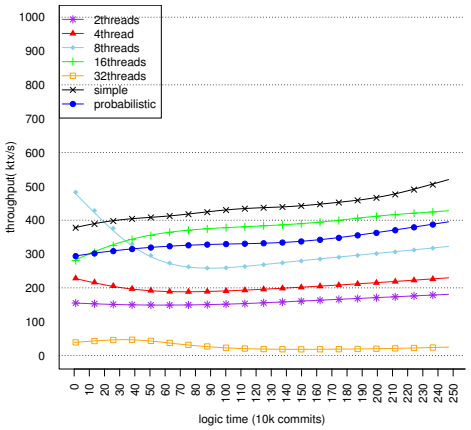
(b) **ssca2**



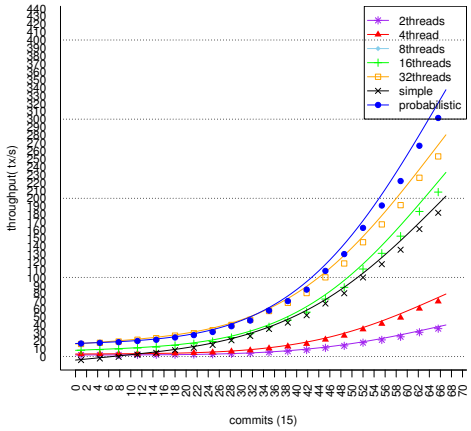
(c) **genome**



(d) **vacation**



(e) **yada**



(f) **labyrinth**

Figure B.2: Throughput comparison for **STAMP**. Regression curves are utilised. Fig. B.2(c) illustrates the original data.

Appendix C

Résumé de Thèse en Français

C.1 Titre de Thèse

Contrôle Autonomique du Parallélisme et du Placement de Threads pour les Mémoires Transactionnelles Logicielles

C.2 Résumé de Thèse

L'exécution de programmes paralléles demande à établir un compromis entre le temps de calcul (nombre de threads) et le temps de synchronisation. Ce compromis dépend principalement du nombre de threads actifs. Un haut degré de parallélisme (beaucoup de threads) permet généralement de diminuer le temps de calcul, mais peut aussi avoir pour conséquence d'augmenter les surcoûts de synchronisation entre threads. De plus, le placement des threads sur les cœurs peut impacter les performances du programme, car le temps pour accéder aux données en mémoire peut varier d'un cœur à l'autre en raison de la contention sur la hiérarchie mémoire. Ainsi, la performance d'un programme peut être améliorée en adaptant le nombre de threads actifs et en plaçant correctement les threads sur les cœurs de calcul. Cependant, il n'existe pas de règle universelle permettant de décider a priori du niveau de parallélisme optimal et du placement de threads d'un programme, en particulier pour un programme avec les changemets de comportement dynamique. D'ailleurs, un paramétrage hors ligne est moins précis. Cette thèse présente un travail sur la gestion dynamique du parallélisme et du placement de threads. Cette thèse s'attaque au problème de gestion de threads utilisant de la mémoire transactionnelle logicielle (Software Transactional Memory, STM). La mémoire transactionnelle logicielle constitue une technique prometteuse pour traiter le problème de synchronisation en évitant les verrous. Le concept de calcul autonomique offre aux programmeurs un cadre de méthodes et techniques pour construire des systèmes auto-adaptatifs ayant un comportement maîtrisé. L'idée clé est d'implémenter des boucles de rétroaction afin de concevoir des contrôleurs sûrs, efficaces et prédictibles, permettant d'observer et d'ajuster de manière dynamique les systèmes contrôlés, tout en minimisant le surcoût d'une telle méthode. La thèse propose de concevoir des boucles de rétroaction afin d'automatiser le gestion de threads à l'exécution avec comme objectif la réduction du temps d'exécution des programmes.